

决策树系列

决策树简介

决策树算法原理

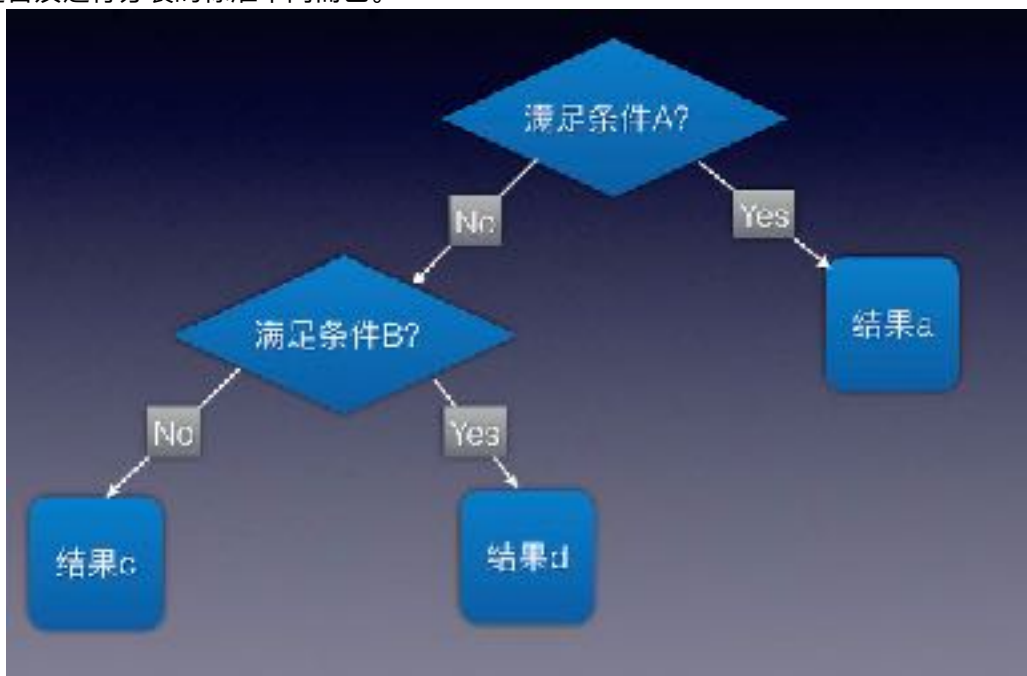
决策树的优点

决策树的缺点

1.1 决策树简介

决策树 (decision tree) 的发展路径是非常的丰富的，除了常见的id3, c4.5, cart之外还有c5.0、chaid、多变量决策树等等，但是考虑到这些算法都是相对比较冷门的，个人感觉暂时没有太多的必要深入介绍，所以后续仅介绍我们常见的也是面试常常问到的id3、c4.5和cart。

决策树可以算是机器学习算法中最直观、最容易解释的简单算法，因为本质就是一大堆if else的逻辑的集合，例如某个人大于XX岁，年收入小于XX万元等，只不过不同的tree其判定是否及进行分裂的标准不同而已。



下面我们就从3个算法的角度进行说明。

1.2 决策树算法原理

我们按照时间顺序，从id3开始谈起。

在了解id3之前，我们需要知道一个叫做“信息熵”的概念，信息熵实际上是来源于信息论的概念，如果大学学习的是通信工程，应该对《信息论》这门课不会陌生，我们平常常说，“这篇文章的信息量很大”，但是有多大呢？信息的量是一个很抽象的概念，如果我们要用数值衡量其具体的大小是比较困难的，直到香农提出了“信息熵”的概念，才解决了对信息进行量化度量的问题——一言以蔽之：信息熵用于衡量信息的不确定性（也有解释为衡量混沌程度的，etc，实际上意思是类似的），例如我们说“这次考试学渣会及格”，则这句话的信息量是很大的，因为学渣是否能及格是一个很大的不确定事件，可能要根据改卷老师的心情、学渣考前突击是否蒙对题或者学渣的卷面字迹是否让改卷人赏心悦目，而“这次靠是学霸会及格”就是一个信息量很小的事件了，除非学霸疯了，乱答，才可能会出现不及格的情况。

（当然这里仅仅是举个例子来描述所谓信息量给人类带来的直观感受，如果要使用信息熵的概念来衡量日常事件实际上是非常局限的，例如考试这种事情，影响因素是很多的，要去完全精确的衡量上述事件的信息熵还是比较困难的。）

那么现在来看看信息熵长什么样：

$$\text{Ent}(D) = - \sum_{k=1}^{|D|} p_k \log_2 p_k$$

式中对数一般取2为底，单位为比特。但是，也可以取其它对数底，采用其它相应的单位，它们间可用换底公式换算。

这里的p表示事件的发生概率，注意，以机器学习中的二分类问题为例，这里的概率p表示的就是是否发生的概率。

为了更加形象的理解，我们直接以id3算法为例进行进一步的阐述：

假设目前一个班有100人，高数考试挂了40个人，挂科的人标签为1，否则为0，此时 $p_1=0.4$ ， $p_0=0.6$ ，则根据信息熵公式

$H(u)=0.9709505944546686$ bit,

现在假设我们存在某一个维度的特征“此人是否平常好好学习”将原始的100个人划分为了50和50人的两堆，50个人不好好学习，50个人平常好好学习，发现50个人里面有10个人是及格的，40个都是不及格的，而50个人好好学习的都及格了，此时我们分别计算这两拨人的信息熵：

不好好学习阵营：

$H(u) = -(0.8 \cdot \log_2(0.8) + 0.2 \cdot \log_2(0.2)) = 0.7219280948873623$

好好学习阵营：

$H(u) = -(1 \cdot \log_2(1) + 0.00001 \cdot \log_2(0.00001))$ (避免计算无意义对数值进行了一定的平滑操作)
 $= 0.00016609640474436814$

可以看到，根据“平常是否好好学习”进行了人群的重新划分之后，不同人群的信息熵——即不确定程度都大大降低，直观上的感受就是不同人群更加的“纯净”了。

现在我们计算一下信息熵前后的差距：

不好好学习阵营：

$\Delta h(u) = 0.9709505944546686 - 0.7219280948873623 = 0.24902249956730627$

好好学习阵营：

$\Delta h(u) = 0.9707844980499242$

则信息熵前后差距的加权总和为：

$0.9707844980499242 / 0.9709505944546686 + 0.24902249956730627 / 0.9709505944546686$

=1.256301818634069

上述的计算结果我们称之为信息增益，其公式为：

$$\text{Gain}(D, a) = \text{Ent}(D) - \sum_{v=1}^V \frac{|D^v|}{|D|} \text{Ent}(D^v) .$$

其中D表示按照某个特征分裂之后得到的某个人群，V则表示分为几个人群。注意要进行加权

那么id3是怎么处理的呢？实际上，我们上述仅仅是举了一个简单的例子，现在我们加入一个新的特征：“该学生上课坐的位置”，取值有“前排”，“中间”，“后排”，现在我们进行划分后，假设划分为 前排：40，中间：30，后排：30，并且前排不及格的人有10个，中间不及格的有10个，后排不及格的有20个，然后我们计算一下信息熵：

前排阵营：

$H(u)=0.5623351446188083$

中间阵营：

$H(u)=0.6365141682948128$

后排阵营：

$H(u)=0.6365141682948128$

则总的信息增益为：

$0.9707844980499242 * 3 - 0.6365141682948128 * 2 + 0.5623351446188083 = 2.201660302178955$

可以看到，根据后面的这个特征进行分裂的信息增益要大于前面的那个特征，所以我们优先选择后面的这个特征进行分裂，这样，id3的原理就很简单的出来了，就是遍历每个特征，计算分裂的增益，然后取最大的信息增益的特征进行分裂。

上述的信息增益的计算过程可以表示为：

信息增益 = 信息熵 - 条件熵：

$$\text{Gain}(D, A) = H(D) - H(D|A)$$

信息增益越大表示使用特征 A 来划分所获得的“纯度提升越大”。

关于id3需要注意的就是：

1.id3是多叉树，效率较低，并且只能处理离散特征，从上面的例子就可以看出来了，信息熵和信息增益都是针对离散特征才能计算的，对于连续特征的计算不是不可以，只不过非常麻烦，比如有一个连续特征是1，2，3。。。100，按照id3的思路就划分出100个分支了，因此针对于连续特征的问题后面的算法进行了改进；

2. id3没有考虑缺失值问题，这个问题后面的tree算法也进行了改进；
3. 信息增益的衡量方式非常容易偏向取值数量特别多的特征，因为取值越多的特征，对于id3来说，会切分的越细，即每个分支的数据越少，一般情况下，每一个节点的“纯度”会越高，整体的信息增益越大。
4. id3时代并没有引入剪枝的概念，对于过拟合没有什么应对方法；
5. 信息熵的计算比较涉及到求和和对数变换，比较费时
- 6.

C4.5

针对于id3，c4.5进行了很大的改进：

1. 针对信息增益对取值数目多的特征有偏好的问题，使用信息增益率替代信息增益，

$$\text{Ent}(D) = - \sum_{k=1}^{|D|} p_k \log_2 p_k .$$

$$\text{Gain}(D, a) = \text{Ent}(D) - \sum_{v=1}^V \frac{|D^v|}{|D|} \text{Ent}(D^v) .$$

回顾一下信息熵和信息增益的公式，

C4.5的做法是：

$$\text{Gain_ratio}(D, a) = \frac{\text{Gain}(D, a)}{\text{IV}(a)}$$

分子部分就是信息增益没有变，分母部分是：

$$IV(a) = - \sum_{v=1}^V \frac{|D^v|}{|D|} \log_2 \frac{|D^v|}{|D|}$$

注意，这里的D表示的是取值的个数，比如某个特征的取值有100个，分裂之后某一枝有50个，则 $|D^v|=50$ ， $|D|=100$ 。实际上就是针对于取值数目多的特征增加了一个惩罚项。

但是信息增益率存在的问题在于，会对取值很少的特征有所偏好，举个极端的例子，假设某个特征取值完全相同，则分母计算结果为0，则信息增益率为无穷大。

因此针对于这个问题，c4.5使用了一种启发式的方法：

先从候选划分特征中找到信息增益高于平均值的特征，再从中选择增益率最高的。

2.针对id3无法处理离散特征的问题，c4.5采用的策略是永远切分为两段，即假设某个连续特征的取值为1~10，则c4.5会遍历连续值的每一个中间间隔，也就是1.5,2.5,3.5.....9.5，分别计算以该划分点作为二元分类点时的信息增益率，并选择信息增益最大的点作为该连续特征的二元离散分类点；需要注意的是，如果当前节点划分选择的特征为连续特征，则该属性可以继续作为后代节点的划分特征。

3.针对缺失值的问题，引入了一种解决方法：

在计算增益率从而选择最优特征进行分裂的时候，信息增益率按照非缺失样本先进行计算，然后进行比重的折算，比如某个特征的信息增

益率为5，但其缺失了50%的数据，则其最终折算后的信息增益率为
 $5 \times 0.5 = 2.5$

假设这个特征存在缺失但是仍旧被选为分裂特征进行分裂，则引入样本权重。举个例子，假设我们在特征a上分裂，特征a缺失样本100个，未缺失样本500个，则现在500个样本上正常分裂，假设分为两枝，一枝300个样本，一枝200个样本，则缺失样本进行权重分配，100个缺失样本分别进入两枝，只不过进入左边的100个样本其样本权重未3/5，右边的100个样本权重为2/5（和不均衡学习中样本赋权的概念是类似的）。

C4.5的缺点：

- C4.5和id3一样用的是多叉树，效率较低，用二叉树效率更高；
- C4.5 和id3分别使用信息增益率和信息增益进行分裂指导，因此只能用于分类问题；
- C4.5 的信息增益率计算和信息熵一样都比较计算复杂而麻烦

Cart

Cart的改进：

- 1.Cart摒弃了麻烦的多叉树，而使用二叉树进行替代，但这也意味着cart无法直接处理类别特征；
- 2.Cart使用了gini指数作为分裂标准；

基尼指数代表了模型的不纯度，基尼系数越小，不纯度越低，特征越好。这和信息增益（率）正好相反。

$$\begin{aligned} \text{Gini}(D) &= \sum_{k=1}^{|Y|} \sum_{k' \neq k} p_k p_{k'} \\ &= 1 - \sum_{k=1}^{|Y|} p_k^2 \end{aligned}$$

基尼指数反映了从数据集中随机抽取两个样本，其类别标记不一致的概率。因此基尼指数越小，则数据集纯度越高。基尼指数偏向于特征值较多的特征，类似信息增益。

3、cart tree分裂不再是计算所谓的衡量标准的增益了，而是

$$\text{Gini_index}(D, a) = \sum_{v=1}^V \frac{|D^v|}{|D|} \text{Gini}(D^v) .$$

直接使用，分裂之后的左右两枝的gini值之和。

剪枝策略

首先介绍一些tree的剪枝策略，第一种，预剪枝，实际上我们通过tree的最大深度或者叶节点的最小样本数量就可以达到预剪枝的效果，除此之外，我们更熟悉的预剪枝是通过验证集的方式来进行的，其过程类似于gbdt或者是nn的early stopping。也就是每一次进行分裂，我们都使用分裂前的tree对验证集进行验证测试其精度，并且使用分裂之后的tree也对验证集测试其精度，二者进行比较从而决定是否进行分裂。

这种剪枝的方法存在的问题在于可能欠拟合，假设根据某个特征a进行分裂之后验证集的精度虽然下降，但是对a进行分裂后又对b进行分裂精度大幅度提升，这种情况下使用预剪枝的方法就造成了较大的欠拟合风险。

实际上这种预剪枝方法所带来的负面效应和对tree的参数进行限制一样都会带来欠拟合的风险，例如对于lightgbm，其leaf-wise的生长方式，对于max_depth进行太过严苛的限制很容易带来“虽然过拟合的情况缓解了，但是我的交叉验证的评价指标结果变差了”的情况。

2、后剪枝

西瓜书上描述的后剪枝策略的思路比较简单，也是引入验证集，对每一个叶子节点的父节点进行删除，然后观察验证集上的精度变化，例如下图：



去掉纹理这个叶子节点的父节点后：



验证集的精度梯度提高，因此进行剪枝

除此之外，还有代价复杂性剪枝、最小误差剪枝、悲观误差剪枝等，因为目前剪枝的策略使用的比较少（基本是用集成tree来处理问题，集成框架下，单棵tree的性能影响不大），c4.5使用的悲观剪枝，而cart使用代价复杂性剪枝叶。

决策树对缺失值的处理：

- 1.c4.5的带权；
- 2.缺失值差补技术；
- 3.Cart的代理分裂（较为复杂未深入研究，大概的思路就是找一个分裂效果最相似又不缺失的特征进行分裂，计算非常复杂）

决策树的优点

决策树的缺点

这个上面貌似都提到过了。。。

另外就是单决策树的泛化性能有限, 不进行剪枝或者是参数限制其复杂度非常容易过拟合;

Adaboost 系列

Adaboost简介

Boosting算法大体上分为两个流派, 一个是以adaboost为代表的boosting框架, 一类是以gbm为代表的boosting框架, 和 AdaBoost 一样, gbm 也是重复选择一个表现一般的模型并且每次基于先前模型的表现进行调整, 不同的是, AdaBoost家族, 包括了离散adaboost, real adaboost, gentle adaboost, logitboost都是 是通过对样本进行reweight来定位模型的不足而 GBM 是通过算梯度 (gradient) 来定位模型的不足, 因此相比 AdaBoost, Gradient Boosting 可以使用更多种类的目标函数。

Adaboost算法原理

Discrete AdaBoost [Freund and Schapire (1996b)]

1. Start with weights $w_i = 1/N, i = 1, \dots, N$.
2. Repeat for $m = 1, 2, \dots, M$:
 - (a) Fit the classifier $f_m(x) \in \{-1, 1\}$ using weights w_i on the training data.
 - (b) Compute $\text{err}_m = E_w[1_{(y \neq f_m(x))}]$, $c_m = \log((1 - \text{err}_m)/\text{err}_m)$.
 - (c) Set $w_i \leftarrow w_i \exp\{c_m 1_{(y_i \neq f_m(x))}\}$, $i = 1, 2, \dots, N$, and renormalize so that $\sum_i w_i = 1$.
3. Output the classifier $\text{sign}[\sum_{m=1}^M c_m f_m(x)]$.

知乎 @马东什么

这里展开说下吧, adaboost和gbdt一样, 整个算法可以用简单的公式进行表示:

$$f(x) = \sum_{i=1}^M \alpha_m G_m(x) \quad (3.1.1)$$

也就是我们常说的“加性模型”，然后我们写出adaboost的目标函数的形式：

$$\min \sum_{i=1}^N L(y_i, f_{m-1}(x) + \alpha_m G_m(x))$$

即求 α_m 是什么

这是前向分布法的目标函数的形式，其中：

$$f_m(x) = f_{m-1}(x) + \alpha_m G_m(x)$$

前向分步算法就通过不断迭代求得了从 到 的所有基分类器及其权重，问题得到了解决。

那么adaboost和指数损失函数是什么关系呢？首先我们需要知道，指数损失函数的地位和二元的logloss损失函数是一样的，他们都是作为0-1损失函数的代理函数来代替原始问题的目标函数的：

$$L(y, f(x)) = \exp(-yf(x))$$

我们将其带入adaboost的目标函数的式子可以得到：

$$\operatorname{argmin}_{\alpha_m, G_m} \sum_{i=1}^N \exp[-y_i (f_{m-1}(x) + \alpha_m G_m(x))]$$

令

$$w_{m,i} = \exp[-y_i f_{m-1}(x)]$$

我们知道：

$$y_i f_{m-1}(x)$$

是前面m-1轮的基学习器的求和与 y_i 的乘积，在本轮即第m轮的计算过程中，前面的m-1个基学习器已经是固定的了（例如tree的深度、tree的叶子节点以及叶子节点值等都已经都是固定的）， y_i 表示实际的标签值也是固定的，因此，常量对于优化问题来说是可以忽略的，所以最终原始的目标函数可以转化为：

$$(\alpha_m, G_m(x)) = \arg \min_{\alpha, G} \sum_{i=1}^N w_{mi} e^{-y_i \alpha G(x_i)}$$

这里， y_i 是-1或者1（注意，离散adaboost的标签是和svm一样设为1或者-1的）， $G(x_i)$ ，即第m轮的基学习器的输出，离散adaboost的基学习器是卡阈值直接输出-1或1的。

这个时候，我们需要求解两个变量：

$$(\alpha_m, G_m(x))$$

离散adaboost采用的是分布求解的策略，即先固定 α ，求解 G ，然后求解出 G 之后再求解 α 。

此时，我们就可以将上式继续转化：

$$\hat{G}_m(x) = \argmin_{G_m} \sum_{i=1}^N w_{m,i} I(y_i \neq G_m(x_i))$$

这里的转化就很有灵性了，针对于：

$$\arg \min_{\alpha, G} \sum_{i=1}^N w_{mi} e^{-y_i \alpha G(x_i)}$$

固定并且 y_i 和 $G(x_i)$ 输出均为1或者-1的情况下，上述的两个最优化的函数时等价的，因为其最优值都是在 $y_i = G(x_i) = 1$ 或-1的情况下取到的。那么此时我们的目标函数就是：

$$\hat{G}_m(x) = \underset{G_m}{\operatorname{argmin}} \sum_{i=1}^N w_{m,i} I(y_i \neq G_m(x_i))$$

这个问题如何求解呢，实际上这个时候的 $G_m(x_i)$ 就是一个基学习器正常训练的过程，具体的过程是，我们在当前的权重和样本上，正常fit一个cart tree，仅此而已，cart tree的拟合过程和上述的目标函数是无关的。

这样，我们就求解出了 $G_m(x_i)$ 了。（这里为了方便省事， $G_m(x_i)$, $G_m(x)$, $G(x_i)$, $G(x)$ 都是一个意思。。懒得改了，_也是一样的处理）

此时我们又回到了当初的起点：

$$(\alpha_m, G_m(x)) = \underset{\alpha, G}{\operatorname{argmin}} \sum_{i=1}^N w_{m,i} e^{-y_i \alpha G(x_i)}$$

只不过这个时候 $G_m(x)$ 已经求解出来了，就好像分手之后你前女友嫁人了一样是没法改变的事实了。此时我们的重心就放在了_的求解上了，对上式进行展开：

$$\sum_{i=1}^N w_{m,i} \exp(-y_i \alpha_m G_m(x)) = \sum_{y_i = G_m(x_i)} w_{m,i} e^{-\alpha} + \sum_{y_i \neq G_m(x_i)} w_{m,i} e^{\alpha}$$

注意，这里

$$= \sum_{y_i = G_m(x_i)} w_{m,i} e^{-\alpha} + \sum_{y_i \neq G_m(x_i)} w_{m,i} e^{\alpha}$$

这里是把分段函数合并成一个了，当 $y_i=G(x_i)$ 的时候只有第一个式子参与了计算，当 $y_i \neq G(x_i)$ 的时候只有第二个式子参与了计算，这一点要注意清楚。

$$(e^\alpha - e^{-\alpha}) \sum_{i=1}^N w_{m,i} I(y_i \neq G_m(x_i)) - e^{-\alpha} \sum_{i=1}^N w_{m,i} = 0$$

这一步的推导，这个挺简单的但是说起来很麻烦，就是构造新式子，懒得写了。。自行体会吧，然后就是求导了，可以得到：

$$\hat{\alpha}_m = \frac{1}{2} \log \frac{1 - e_m}{e_m}$$

其中 e_m 是带权分类误差率：

$$e_m = \frac{\sum_{i=1}^N w_{m,i} I(y_i \neq G_m(x_i))}{\sum_{i=1}^N w_{m,i}}$$

注意啊，这里样本是带权重的和我们常规的所有样本等权计算是不一样的。并且分母加入了

$$\sum_{i=1}^N w_{m,i}$$

是希望整体的样本权重为1。

这里我们贴出这篇文章的作者写的ppt上的演示图：

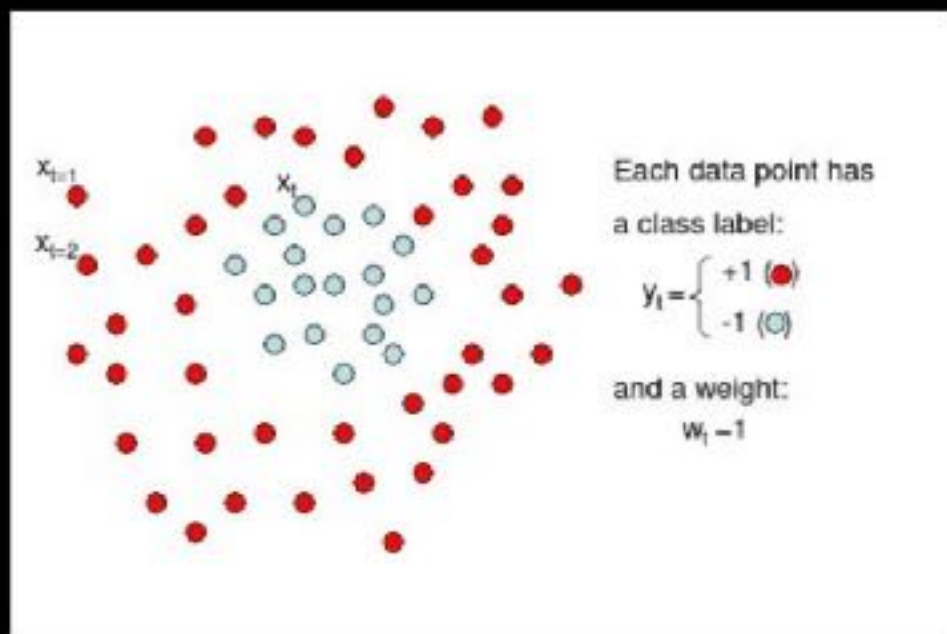


Figure: A toy example of boosting. It is a sequential procedure. 知乎 @马东什么

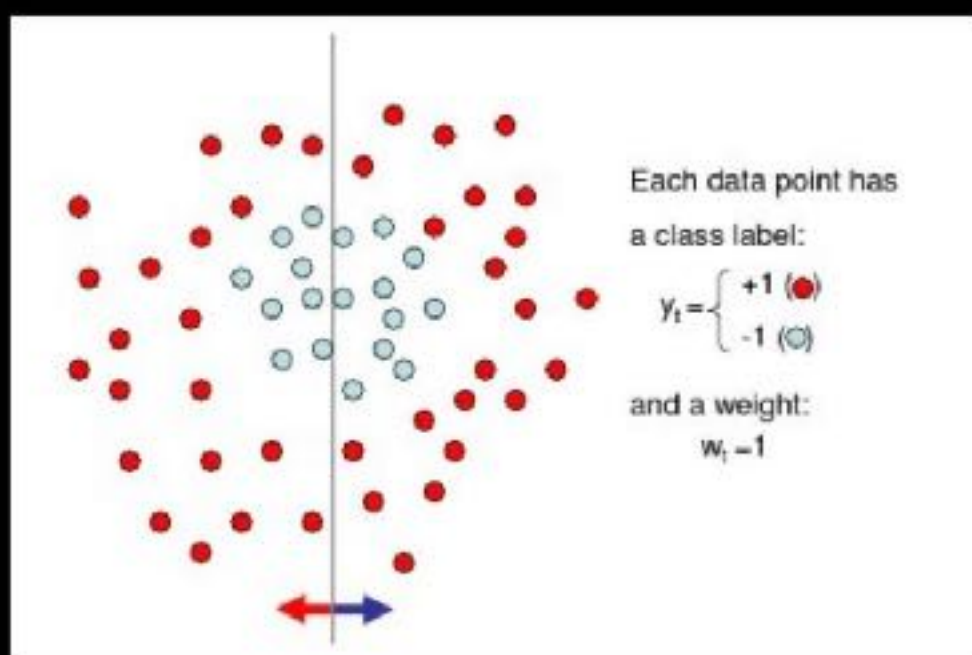


Figure: Random guessing: $err_m = 0.5$. 知乎 @马东什么

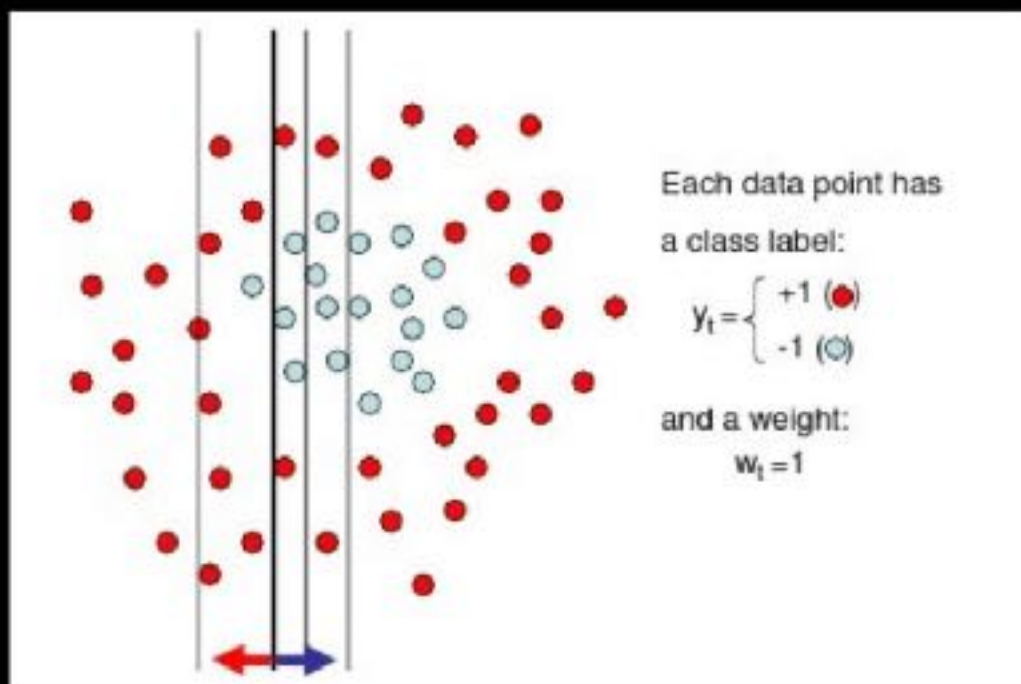


Figure: A weak classifier with lowest error rate: [知乎 @马东什么](#)

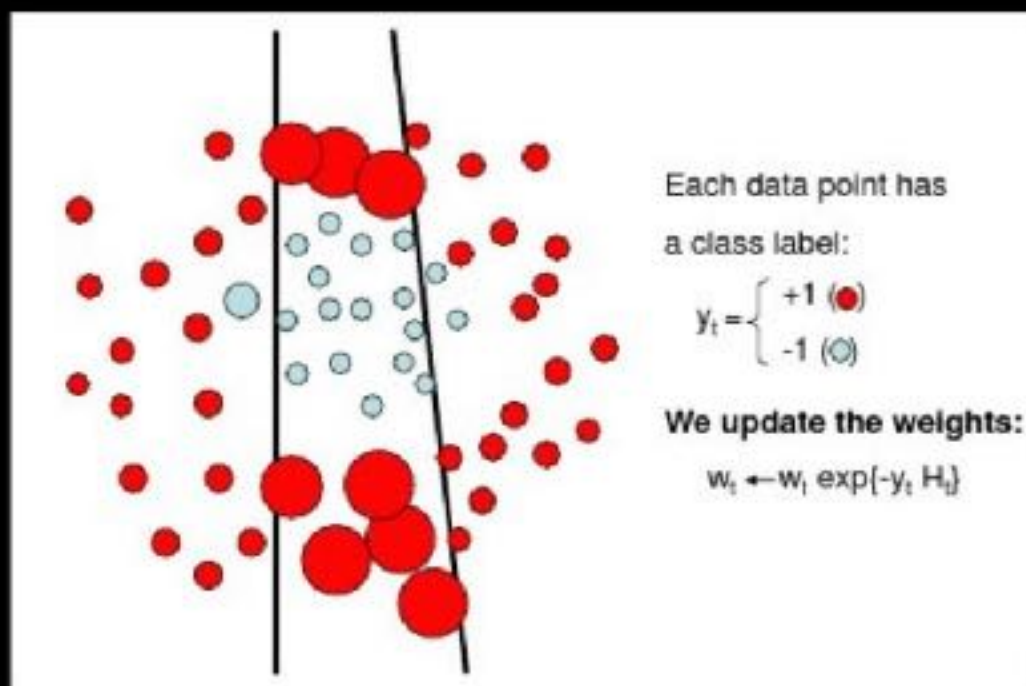
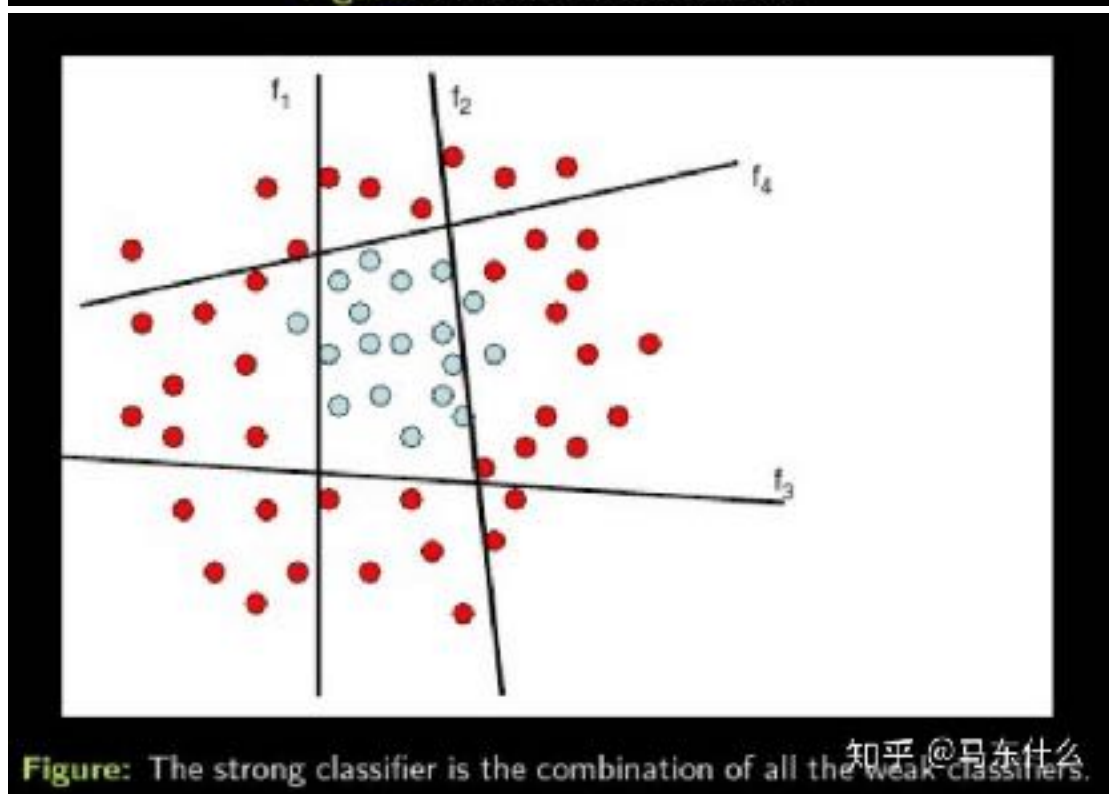
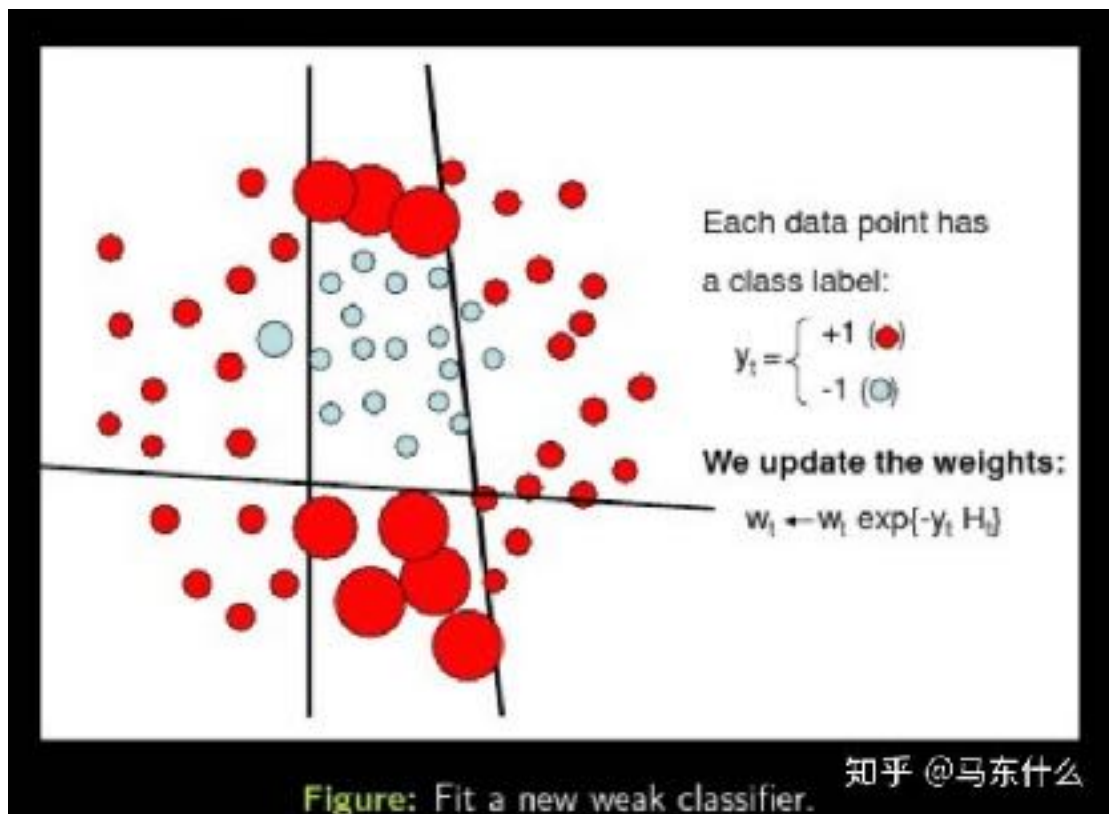
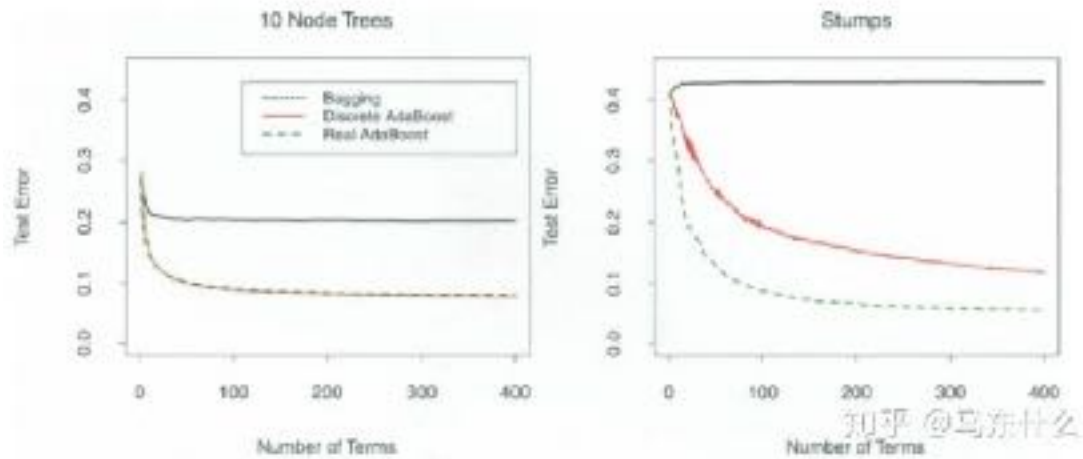


Figure: Reweight training samples. [知乎 @马东什么](#)



这里作者为了画图方便用了决策树状，因此每一个基cart的分界面都是一条线，实际上如果树的深度较大的化分界面是一条折线的。



这里作者做了一个测试，左边这个图不用注意主要是为了让你知道每条线代表了啥，右边才是需要注意的，stumps使用的是决策树桩，可以看到bagging对决策树状进行集成效果是很渣的，bagging框架只能减少方差而无法降低偏差，因此对于基学习器的拟合能力是有要求的，但是adaboost就很神奇了，**不仅能降低偏差还能降低方差！**降低偏差好理解，每一轮的基学习器都是拟合上一轮的预测偏差，那么为啥adaboost还能降低方差呢？

主要原因就在于，adaboost每一轮的样本权重都是在变化的，实际上使用的是广义的采样方案。我们知道**bagging**每一轮是随机选择部分样本进行训练的，实际上这等同于每一轮对部分样本赋权为0，部分样本赋权为1。

而adaboost则是通过对每一样本进行赋权从而达到了bagging的效果。因此，实际上boosting的框架是既能起到减少方差的效果也能起到降低偏差的效果的。。

上图我们还看到，有一个叫做real adaboost的东西。我们前面提到过离散型的adaboost每一轮直接输出分类标签，因此需要我们去卡不同的分类阈值，大佬不甘寂寞，又想到了一种更好的处理方法：

Discrete AdaBoost [Freund and Schapire (1996b)]

1. Start with weights $w_i = 1/N, i = 1, \dots, N$.
2. Repeat for $m = 1, 2, \dots, M$:
 - (a) Fit the classifier $f_m(x) \in \{-1, 1\}$ using weights w_i on the training data.
 - (b) Compute $\text{err}_m = E_w[1_{(y \neq f_m(x))}]$, $c_m = \log((1 - \text{err}_m)/\text{err}_m)$.
 - (c) Set $w_i \leftarrow w_i \exp[c_m 1_{(y_i \neq f_m(x_i))}]$, $i = 1, 2, \dots, N$, and renormalize so that $\sum_i w_i = 1$.
3. Output the classifier $\text{sign}[\sum_{m=1}^M c_m f_m(x)]$.

知乎 @马东什么

我们之前求解两个参数的时候，和G的时候，是先固定了，也就是上面的伪代码中的Cm，先训练了G，然后再去求Cm的。如下图

离散AdaBoost

离散adaboost采用的是分布求解的策略，即先固定 α ，求解G，然后求解出G之后再求解 α 。

此时，我们就可以将上式继续转化：

$$\hat{G}_m(x) = \underset{G_m}{\operatorname{argmin}} \sum_{i=1}^N w_{m,i} I(y_i \neq G_m(x_i))$$

这里的转化就很有灵性了，针对于：

$$\underset{\alpha, G}{\operatorname{argmin}} \sum_{i=1}^N w_{m,i} e^{-y_i \alpha G(x_i)}$$

α 固定并且 y 和 $G(x)$ 输出均为1或者-1的情况下，上述的两个最优化的函数是等价的，因为其最优值都是在 $y = G(x) = 1$ 或 -1 的情况下取到的。那么此时我们的目标函数就是：

上面是离散adaboost，下面是概率型adaboost，我们做一下比较；

Real AdaBoost

1. Start with weights $w_i = 1/N, i = 1, 2, \dots, N$.
 2. Repeat for $m = 1, 2, \dots, M$:
 - (a) Fit the classifier to obtain a class probability estimate $p_m(x) = \hat{P}_m(y = 1|x) \in [0, 1]$, using weights w_i on the training data.
 - (b) Set $f_m(x) \leftarrow \frac{1}{2} \log p_m(x)/(1 - p_m(x)) \in R$.
 - (c) Set $w_i \leftarrow w_i \exp[-y_i f_m(x_i)], i = 1, 2, \dots, N$, and renormalize so that $\sum_i w_i = 1$.
 3. Output the classifier $\text{sign}[\sum_{m=1}^M f_m(x)]$.
-

ALGORITHM 2. The Real AdaBoost algorithm uses class probability estimates $p_m(x)$ to construct real-valued contributions $f_m(x)$.

而real adaboost则使用了另一种方式：

- 1、标签换成了我们熟悉的0, 1而不是-1和1；
- 2、G不再是简单的卡阈值输出1或者-1，而是输出

$$f_m(x) \leftarrow \frac{1}{2} \log p_m(x)/(1 - p_m(x)) \in R.$$

这里的fm(x)对应的就是我们的G了；这里pm(x)是G输出的样本的概率值。。。其实就是输出两种类别预测概率的比值

然后进行了下一步的权重更新：

$$(c) \text{ Set } w_i \leftarrow w_i \exp[-y_i f_m(x_i)], i = 1, 2, \dots, N, \text{ and renormalize so that } \sum_i w_i = 1.$$

具体的推导可见原论文...太麻烦了就不写了

Adaboost的优点

- 1) Adaboost 的拟合能力强

2) 提供了集成的框架，基学习器可以灵活选择，当然考虑到性能、多样性等问题目前还是以tree为主；

Adaboost的缺点

- 1) 集成学习的训练时间都很长
- 2) 对hard sample 敏感，异常样本在迭代中可能会获得较高的权重，影响最终的强学习器的预测准确性。
- 3) 执行效果依赖于弱分类器的选择

GBDT 系列

gbdt简介

Gbdt算法原理

Gbdt的优点

Gbdt的缺点

1.1 gbdt简介

Gbdt，全称：Gradient Boosting Decision Tree，实际上是基于GBM（gradient boosting machine）集成学习框架下的一种非常流行的应用，就好比我们常说的bagging框架，二者的基学习器，理论上可以使用大部分常见的有监督模型。

从偏差-方差均衡理论的角度来说，我们希望模型能够有低方差和低偏差，但在集成学习的思想出来之前，人们普遍认为二者就像事业和爱情一样难以兼得。直到bagging gbm stacking等集成思想被提出之后，人们发现神奇的事情，bagging可以在保持偏差变动小的情况下有效地降低方差，而最初的gbm 主要在于降低偏差，后续引入了rf的思想，加入了行列采样，神奇的使得整个框架能够达到同时降低方差和偏差的效果。

集成学习的最终奥义：“好而不同”，而gbm框架本身可以较好地解决偏差和方差问题，因此，对于基学习器的精度和泛化性能并没有太过高昂地要求，质量不够，数量来凑，但有一个问题，基学习器必须具有多样性，如果基学习器都是非常接近的，神仙也白搭（这也是为什么xgb、lgb、catboost进行集成效果往往比较有限，因为很多时候确实差别并不大），所以，我们就需要根据目前的情况选择合适的基学习器——tree和nn，这二者的多样性是很好的，在不同的子数据集上，tree和nn的模型最终的训练结果是非常多样化的，以tree为例，在采样之后不同的数据子集上得到的树的结构差异性很大。之所以nn并没有成

为GBM框架下主流的基学习器，主要的原因是tree的训练速度快，因为不涉及到nn复杂繁琐的多epochs的迭代。

1.2 gbdt算法原理

这里我们先介绍最初的gbdt的原理部分，理解一个算法前，我们如果能够较好的形象的以公式的形式对整个模型进行表示，然后又能够知道其损失函数，就容易理解的多，见下：

$$f(x) = f_0(x) + \sum_{t=1}^T \text{learningrate} * f_t(x)$$

(这是整个模型的公式化)

$$Obj = \sum_{i=1}^n l(y_i, \hat{y}_i)$$

(这是模型的目标函数，即损失函数)

最初的gbdt的设计并不复杂， $f_0(x)$ 表示初始的基学习器，后面那一项表示后续拟合的所有其它基学习器，实际上早期的gbdt的思想是非常直观的，以回归问题为例，当我们在原始数据上训练得到第一颗tree之后，我们可以计算出真实的label和预测的label之间的差异。

假设我们原始的label是：10.8,5.0,4.3,2.5,8.9，而我们预测出的label是：10.5,5.3,4.4,2.6,9.0，我们计算一下二者的差值，看看当前的模型的预测误差是怎么样：

$$0.3, -0.3, -0.1, -0.1, -0.1$$

这个时候，一个很直观的感觉出来了，既然原始的tree存在误差，那么我们不是可以以原始的特征矩阵为特征，误差作为新的标签来训练一颗新tree，然后直接对二者进行求和，不就能神奇的降低误差了吗？我们上述的计算结果称之为残差，注意，残差的计算方向是——真实标签-预测结果，而我们前面的计算，实际上就是平方损失函数的负梯度：

$$L(y, F(x)) = \frac{1}{2}(y - F(x))^2$$

上式为平方损失函数，我们对其进行求导可得：

$$\frac{\partial L}{\partial F(x_i)} = \frac{\partial \sum_i l(y_i, F(x_i))}{\partial F(x_i)} = \frac{\partial l(y_i, F(x_i))}{\partial F(x_i)} = F(x_i) - y_i$$

其一阶梯度，取负号就可以得到 $y_i - f(x_i)$ ，也就是我们上面例子里进行的计算。

，这样一来，分类问题也能很好的理解了，例如对于二分类问题，计算二元交叉熵的负梯度，将负梯度作为下一轮的tree进行拟合时使用的标签。

需要补充的是，gbdt引入了所谓shrinkage的正则化手段对模型进行约束，说白了就是引入了学习率，Shrinkage（缩减）的思想认为，每次走一小步逐渐逼近结果的效果，要比每次迈一大步很快逼近结果的方式更容易避免过拟合，这实际上和逻辑回归进行梯度下降法时使用学习率的目的是非常类似的：

1.3 gbdt的优点

1、低偏差，拟合能力强，同时加入了rf的采样的思想后，gbdt在方差和偏差方面都有非常

好的表现，是目前在各个领域的一种非常常用和流行的算法，尤其是结构化数据的领域，例如典型的风控领域，gbdt可以说是统治了半壁江山。

- 1.在训练的过程中自动进行特征选择，将上游的特征工程和下游的模型训练灵活的结合在一个训练过程中；
- 2.能较好的处理非线性问题；
- 3.能够适应多种损失函数，解决不同情况下的问题（只要你能将实际的问题抽象成一个有监督问题，并且其损失函数是可导的）
- 4.可以非常方便的处理缺失值和异常值的问题（当然，gbdt本身对于缺失和异常值的处理策略并不一定能够给模型带来最好的效果）
- 5.可解释性相对较好，通过feature importance，部份依赖图，shap等等可以较好的解释特征对于预测结果的影响从而得到相对来说人类逻辑上能够解释的结论，shap更是可以从样本粒度上去解释模型对样本的预测原因；
- 6.一般情况下不需要对特征进行归一化处理；
- 7.自动进行特征组合，能够找到有意义的高阶特征组合；

1.4 gbdt的缺点

- 1、难以处理高维稀疏的数据，因为tree本身的正则化对于高维稀疏的数据情况不像l1正则化能够带来有效的约束，并且tree本身的分裂在稀疏的情况下显著性很差；
- 2、对于异常点（注意是异常点不是异常值）较为敏感，因为gbdt会在因为异常导致预测误差特别大地样本上不断地去用新的tree来拟合，导致模型太过拟合异常样本，最终的结果就是泛化性能差；例如在回归问题中，假设标签是【1.6, 2.6, 3.5, 1.5, 1000】，这种情况下gbdt会不断的用新tree去拟合标签为1000的样本的负梯度。
- 3、集成模型本身的计算复杂度都是比较高的，训练耗时；

Xgboost 系列

Xgboost简介

Xgboost算法原理

Xgboost的优点

Xgboost的缺点

1.1 xgbost简介

XGBoost是Extrem Gradient Boosting（极限梯度提升）的缩写，它是基于决策树的

集成机器学习算法，大体上沿袭了之前说过的gbdt的框架，但是在此之上做了很多的改进：

首先是算法上的：

1.tree的正则化概念的引入：传统的gbdt对模型进行优化的方法就是引入学习率来缓解模型过拟合的问题，而xgboost则引入了树的正则化的方法，这和在线性回归中引入正则化的思路是很类似的，都是在原始的损失函数上加入对模型的复杂度的定义作为损失函数的补充项，只不过线性回归是以权重系数的l1或l2范数作为损失函数的补充项，而xgboost的定义则如下：

$$Obj = \sum_{i=1}^n l(y_i, \hat{y}_i) + \sum_{k=1}^K \Omega(f_k) \quad (\text{这里公式后期和上面统一一下})$$

$$\Omega(f_t) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

其实抛开前面的gamma*T，后面这项的定义式和l2正则化的形式是完全一样的。这里需要注意，T表示叶子节点的数量，w表示叶子节点的权重，也就是叶子节点的值，这里注意一下，其实二者是一个意思，很多地方分开叫了。。。有些地方也叫leaf score）Ti表示叶子节点的数量，gamma则是这一个惩罚项的超参数；

从人类逻辑上来说，如果我们希望gbdt的模型能够尽量符合奥卡姆剃刀原则，那么就希望叶子节点的数目能够少一点，那么第二项如何理解？其实xgboost的最终输出非常类似于一个逻辑回归

$$w_1 x_1 + w_2 x_2 + w_3 x_3 + \dots + w_n x_n$$

这里x1-xn表示n个叶子节点，wi表示叶子节点的权重，xi的取值不是0就是1，1表示样本落入了某棵树的某个叶子节点，0表示没有落入。那么和逻辑回归一样，我们希望权重整体是偏小的，因为过大的权重会给预测结果带来很大的不稳定性，比如w1=10000，而其它的wi均为0.01，显然，模型的决策会在非常大的程度上受到w1的叶子节点的影响从而导致了过拟合的巨大风险。因此，这一惩罚项可以说起到了和逻辑回归中l2正则化非常相似的作用。

2.梯度信息的使用：原始的gbdt仅仅使用了一阶梯度的信息，而xgboost使用了一、二阶梯度，具体是如何实现的呢？Xgboost对损失函数实际上进行了二阶泰勒展开，这里要注意，我们所说的二阶泰勒展开是针对于gbdt原始的损失函数进行展开的，不包括tree的正则项的部分。

$$f(x) = \frac{f(x_0)}{0!} + \frac{f'(x_0)}{1!}(x-x_0) + \frac{f''(x_0)}{2!}(x-x_0)^2 + \dots + \frac{f^{(n)}(x_0)}{n!}(x-x_0)^n + R_n(x)$$

复习一下，这是泰勒展开式，所谓的二阶泰勒展开就是n取2的时候上式的情况。在高数里，泰勒展开式具有很多重要的用途，例如在计算机中，计算机可不会直接求，，，等函数的具体值，通过泰勒公式展开函数可求其近似值，当然，不必太过深入研究，只要

理解这个概念即可，一言以蔽之——泰勒展开式可以用多项式函数去逼近光滑函数。这里xgboost所做的就是直接对原始的损失函数进行二阶泰勒展开，舍弃其误差项，将展开之后的新的损失函数作为模型训练过程中使用的损失函数。

3.缺失值处理：在xgboost的原论文中，其对缺失值的处理被称之为所谓的“稀疏感知”，可能很多人对xgboost的这种处理方式没什么概念，实际上tree对于缺失值的处理在xgboost之前是有很多策略的，例如c4.5的带权的方式，cart的代理分裂等等，但是这类策略对于少量tree和少量数据的训练来说计算复杂度还是可以接受的，但是当样本量很大，特征维度很高，并且tree的数量很多的情况下，就非常的耗费时间了，xgboost对于缺失值的处理可以说是非常简单甚至是有些粗糙的，将缺失样本分别尝试放入左枝和右枝，直接根据增益大小，将缺失样本放入增益最大的分支。

4.引入了随机森林的思想，使用行列采样的方式来进一步缓解过拟合；

5.xgboost的基模型不再是cart tree了，因为分裂的判定条件已经不是看gini了而是根据下面的公式来决定是否进行分裂；

$$Gain = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma$$

6.Xgboost在gbdt的基础上又发展出了dart tree和gblinear；

7.Xgboost使用多种近似分裂方式来大大加速建树的过程；

- exact：精确的贪婪算法。枚举所有拆分的候选人。
- approx：使用分位数草图和梯度直方图的近似贪婪算法。
- hist：更快的直方图优化的近似贪婪算法。
- gpu_hist：GPU hist算法的实现。

8.Xgboost使用

其次是工程上的：

- 块并行（Column Block for Parallel Learning）。在树生成过程中，需要花费大量的时间在特征选择与切分点选择上，并且这部分时间中大部分又花费在了对特征值得排序上。那么怎么样减小这个排序时间开销呢？作者提出通过按特征进行分块并排序，在块里面保存排序后的特征值及对应样本的引用，以便于获取样本的一阶、二阶导数值。具体方式如图：

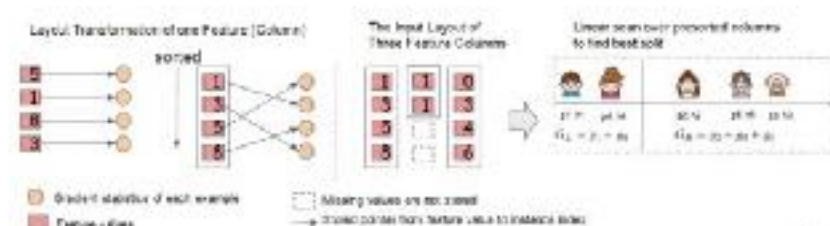


Figure 8: Block structure for parallel learning. Each column in a block is sorted by the corresponding feature value. A linear scan over one column in the block is sufficient to enumerate all the split points.

- 通过顺序访问排序后的块遍历样本特征的特征值，方便进行切分点的查找。此外分块存储后多个特征之间互不干涉，可以使用多线程同时对不同的特征进行切分点查找，即特征的并行化处理。注意到，在顺序访问特征值时，访问的是一块连续的内存空间，但通过特征值持有的索引（样本索引）访问样本获取一阶、二阶导数时，这个访问操作访问的内存空间并不连续，这样可能造成cpu缓存命中率低，影响算法效率。那么怎么解决这个问题呢？缓存访问 Cache-aware Access 。
- 缓存访问（Cache-aware Access）。为了减小非连续内存的访问带来缓存命中率低问题，作者提出了缓存访问优化机制。解决思路是：既然是非连续内存访问带来问题，那么去掉非连续内存访问就可以解决。那么怎么能去掉非连续内存空间的访问呢？转非连续为连续——缓存预取。即提起将要访

问的非连续内存空间中的梯度统计信息（一阶、二阶导数），放置到连续的内存空间中。具体的操作就是为每个线程在内存空间中分配一个连续的buffer缓冲区，将需要的梯度统计信息存放在缓冲区中。这种方式对数据量大的时候很有用，因为大数据量时，不能把所有样本都加入到内存中，因此可以动态的将相关信息加入到内存中。

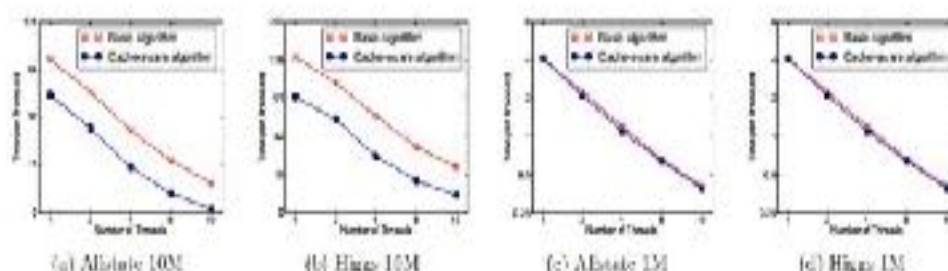
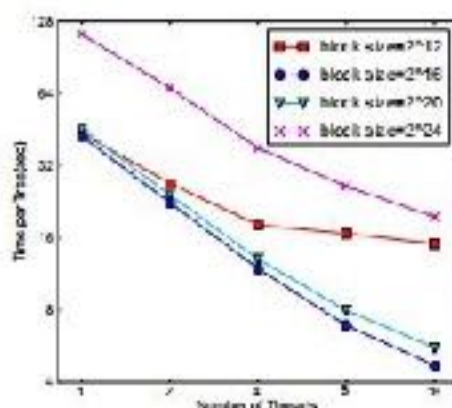
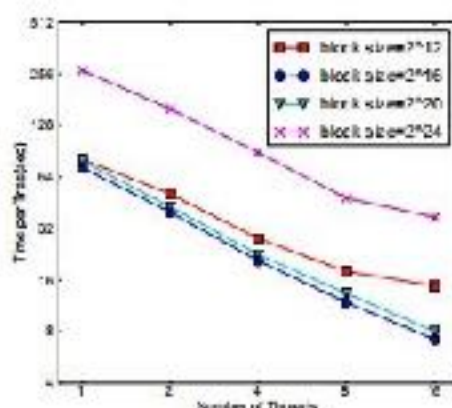


Figure 7: Impact of cache-aware prefetching in exact greedy algorithm. We find that the cache-aware prefetching impacts the performance on the large datasets (10 million instances). Using cache aware prefetching improves the performance by factor of two when the dataset is large.

上图给出了在Higgs数据集上使用缓存访问和不使用缓存访问的式样对比。可以发现在数据量大的时候，基于精确的贪心算法使用缓存预取得处理速度几乎是普通情况下的两倍。那么对于block块应该选择多大才合理呢？作者通过实验证明选择每个块存放 16 个样本时效率最高。



(a) Allstate 10M



(b) Higgs 10M

Figure 9: The impact of block size in the approximate algorithm. We find that overly small blocks results in inefficient parallelization, while overly large blocks also slows down training due to cache misses.

- "核外"块计算 (Blocks for Out-of-core Computation) 。当数据量非常大的时候我们不能把所有数据都加载内存中，因为装不下。那么就必须的将一部分需要加载进内存的数据先存放在硬盘中，当需要时在加载进内存。这样操作具有很明显的瓶颈，即硬盘的io操作速度远远低于内存的处理速度，那么肯定会存在大量等待硬盘io操作的情况。针对这个问题作者提出了“核外”计算的优化方法。具体操作为，将数据集分成多个块存放在硬盘中，使用一个独立的线程专门从硬盘读取数据，加载到内存中，这样算法在内存中处理数据就可以和从硬盘读取数据同时进行。为了加载这个操作过程，作者提出了两种方法。
- 块压缩 (Block Compression) 。论文使用的是按列进行压缩，读取的时候用另外的线程解压。对于行索引，只保存第一个索引值，然后用16位的整数保存与该block第一个索引的差值。作者通过测试在block设置为 个样本大小时，压缩比率几乎达到26% - 29%（貌似没说使用的是什么压缩方法.....）。
- 块分区 (Block Sharding) 。块分区是将特征block分区存放在不同的硬盘上，以此来增加硬盘io的吞吐量。
-
-

1.2 xgboost算法原理

Xgboost的公式表示和gbdt是一样的，只不过训练的过程中存在较多的差异：

$$f(x) = f_0(x) + \sum_{t=1}^T learningrate * f_t(x)$$

这里需要注意的是，无论是xgboost还是lightgbm，初始化的f0(x)都是取一个常量，例如面对二分类问题，初始的f0(x)全部预测为常数（多分类和回归没研究过，感兴趣的可以自己写个自定义损失函数print一下看看）；

<https://zhuanlan.zhihu.com/p/75217528>

那么xgboost是如何训练出来的，首先看下损失函数：

$$L(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \quad (2)$$

$$\text{其中} \Omega(f) = \gamma T - \frac{1}{2} \lambda \|w\|^2$$

我们对上式进行展开，变成下面的式子：表示样本i在t次迭代后的预测值=样本i在前t-1次迭代后的预测值+当前第t颗树预测值。则目标函数可以表示为：

$$\begin{aligned} L^{(t)} &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t)}) + \sum_i \Omega(f_t) \\ &= \sum_{i=1}^n l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t) + constant \quad (3, 4) \end{aligned}$$

注意： $\hat{y}_i^{(t-1)}$ 表示前面所有的t-1棵树总的预测结果，而不是第t-1棵树的预测结果；
式（4）表示贪心地添加使得模型提升最大的 其中constant表示常数项,这个常数项是指前

$$\sum_i \Omega(f_i)$$

t-1次迭代的惩罚项为一个常数，即公式中

部分，在第t次迭代时，前t-1次迭产生的t-1颗树已经完全确定，则t-1颗树的叶节点以及权重都已经确定，所以变成了常数。在式（4）中如果考虑平方损失函数，则式（4）可以表示为：

$$\begin{aligned} L^{(t)} &= \sum_{i=1}^n \left(y_i - (\hat{y}_i^{(t-1)} + f_t(x_i)) \right)^2 + \Omega(f_t) + \text{constant} \\ &= \sum_{i=1}^n \left(y_i - \hat{y}_i^{(t-1)} - f_t(x_i) \right)^2 + \Omega(f_t) + \text{constant} \quad (\text{式5}) \end{aligned}$$

式5中 $y_i - \hat{y}_i^{(t-1)}$ 表示残差，即经过前t-1颗树的预测之后与真实值之间的差距，也就是在GBDT中所使用的残差概念。在XGBoost中提出使用二阶泰勒展开式近似表示式5.泰勒展开式的二阶形式为：

$$f(x + \Delta x) \simeq f(x) + f'(x)\Delta x + \frac{1}{2}f''(x)\Delta x^2 \quad (\text{式6})$$

定义 $g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}), h_i = \partial_{\hat{y}_i^{(t-1)}}^2 l(y_i, \hat{y}_i^{(t-1)})$ 则根据式6可以将式4表示为：

$$L^{(t)} \simeq \sum_{i=1}^n [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) + \text{constant} \quad (\text{式7})$$

注意式7中 $l(y_i, \hat{y}_i^{(t-1)})$ 部分表示前t-1次迭代的损失函数，在当前第t次迭代来说已经是一个确定的常数，省略常数项则得到下面的式子：

$$L^{(t)} = \sum_{i=1}^n [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \quad (\text{式8})$$

则我们的目标函数变成了公式8.可以看出目标函数只依赖于数据点的一阶和二阶导数。其中

$$g_i = \partial_{\hat{y}_i^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)}), h_i = \partial_{\hat{y}_i^{(t-1)}}^2 l(y_i, \hat{y}_i^{(t-1)})$$

中

接下来针对公式8进行细分。首先定义集合 S_j 为树的第j个叶节点上的所有样本点的集合，即给定一颗树，所有按照决策规则被划分到第j个叶节点的样本集合。则根据式2中对模型复杂度惩罚项的定义，将其带入式8有：

$$\begin{aligned}
L^{(t)} &= \sum_{i=1}^n [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t) \\
&= \sum_{i=1}^n [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \\
&= \sum_{j=1}^T [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \quad (式9)
\end{aligned}$$

对式9进行求导：

$$\begin{aligned}
&\Leftrightarrow \frac{\partial L^{(t)}}{\partial w_j} = 0 \\
&\rightarrow (\sum_{i \in I_j} g_i) + (\sum_{i \in I_j} h_i + \lambda) w_j = 0 \\
&\Rightarrow (\sum_{i \in I_j} h_i + \lambda) w_j = - \sum_{i \in I_j} g_i \\
&\Rightarrow w_j^* = - \frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \quad (式10)
\end{aligned}$$

这里就得到了权重w的计算公式了，实际上这里的w对应的概念就是常规的决策树的叶节点的具体的值，比如二分类问题中，cart的某个叶子节点的样本有1000个，900个坏样本，100个好样本，则其对应的叶子节点值——即所谓的叶节点权重w就是900/1000=0.9,而xgboost对于权重的定义比这复杂的多，是通过损失函数推导而出的，式10的意义就是某个叶子节点的值是这个叶子节点的样本的一阶梯度的平方和除以二阶梯度之和加上正则化系数lambda；

将（式10）代入（式9）中得到（式11）：

$$\begin{aligned}
L^{(t)} &= -\frac{1}{2} \sum_{j=1}^T \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T \quad (式11) \\
&\Leftrightarrow G_j = \sum_{i \in I_j} g_i, H_j = \sum_{i \in I_j} h_i \quad \text{则, (式11) 可以简化为 (式12)}
\end{aligned}$$

$$L^{(t)} = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T \quad (式12)$$

到目前我们得到了（式12），可以做为得分值评价一颗树的好坏，那么评价一颗树的好坏有什么用呢？可以用于对的剪枝操作(防止过拟合),和决策树中的剪枝是一样的,给定一个损失函数,判断剪枝后,根据损失函数是否减小来决定是否执行剪枝,只是XGBoost是运用式12来作为损失函数判断的标准。注意到评价一颗树的还好的前提是我们能得到一颗树，上式也是基于给定一个树的前提下推导而来的，那么这颗树怎么来得到呢？

树的生成

在决策树的生成中，我们用ID3、C4.5、Gini指数等指标去选择最优分裂特征、切分点(CART时)，XGBoost同样定义了特征选择和切分点选择的指标：

$$Gain = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma \quad (式13)$$

XGBoost中使用式(式13) 判断切分增益，Gain值越大，说明分裂后能使目标函数减少越多，就越好。其中 $\frac{G_L^2}{H_L + \lambda}$ 表示在某个节点按条件切分后左节点的得分， $\frac{G_R^2}{H_R + \lambda}$ 表示在某个节点按条件切分后右节点的得分， $\frac{(G_L + G_R)^2}{H_L + H_R + \lambda}$ 表示切分前的得分， γ 表示切分后模型复杂度增加量。现在有了判断增益的方法，就需要使用该方法去寻找最优特征以及最优切分点。

因此，实际上xgboost中的base tree使用的是gain所对应的函数进行分裂的指导的，这里实际上实现了预剪枝的功能，并且可以通过gamma来进行灵活的调整，

1.3 xgboost的优点

- 1、低偏差，拟合能力强，同时加入了rf的采样的思想后，gbdt在方差和偏差方面都有非常好的表现，是目前在各个领域的一种非常常用和流行的算法，尤其是结构化数据的领域，例如典型的风控领域，gbdt可以说是统治了半壁江山。
 - 8.在训练的过程中自动进行特征选择，将上游的特征工程和下游的模型训练灵活的结合在一个训练过程中；
 - 9.能较好的处理非线性问题；
 - 10.能够适应多种损失函数，解决不同情况下的问题（只要你能将实际的问题抽象成一个有监督问题，并且其损失函数是可导的）
 - 11.可以非常方便的处理缺失值和异常值的问题（当然，gbdt本身对于缺失和异常值的处理策略并不一定能够给模型带来最好的效果）
 - 12.可解释性相对较好，通过feature importance，部份依赖图，shap等等可以较好的解释特征对于预测结果的影响从而得到相对来说人类逻辑上能够解释的结论，shap更是可以从样本粒度上去解释模型对样本的预测原因；
 - 13.一般情况下不需要对特征进行归一化处理，开箱即用性比较强；
 - 14.自动进行特征组合，能够找到有意义的高阶特征组合；
- 上述的优点和gbdt是一样的，下面主要说下xgboost相对于gbdt的改进：

- 传统GBDT以CART作为基分类器，xgboost还支持线性分类器，这个时候xgboost相当于带L1和L2正则化项的逻辑斯蒂回归（分类问题）或者线性回归（回归问题）。

- xgboost使用了新的dart tree的生成策略，具体的原理可见XXXX

- 传统GBDT在优化时只用到一阶导数信息，xgboost则对代价函数进行了二阶泰勒展开，同时用到了一阶和二阶导数。下面来自xgb的官网叙述：

如果我们考虑使用平方误差 (MSE) 作为损失函数，则目标变为

$$\begin{aligned} \text{obj}^{(t)} &= \sum_{i=1}^n (y_i^{(t-1)} - f(\phi_i)) ^2 + \sum_{j=1}^J \Omega(f_j) \\ &= \sum_{i=1}^n [2(y_i^{(t-1)} - f(\phi_i)) + \eta_i f_j^2] + \text{constant} \end{aligned}$$

可以看到，损失函数为mse的时候，注意，此时我们没有进行二阶泰勒展开

MSE的形式是友好的，具有一阶项（梯度称为残差）和二阶项。对于其他非凸损失（例如，泊松损失），类似地展开好的求解并不容易。因此，在一般情况下，我们考虑对非凸的损失函数进行泰勒展开。

$$\text{obj}^{(t)} = \sum_{i=1}^n \eta_i (y_i^{(t-1)} - g_i / f(\phi_i) + \frac{1}{2} h_i f_{\phi_i}^2(\phi_i)) - \Omega(f_t) + \text{constant}$$

在这里 g_i 和 h_i 被定义为

$$\begin{aligned} g_i &= \partial_{f(\phi_i)} \eta_i (y_i^{(t-1)} - f(\phi_i)) \\ h_i &= \partial_{f(\phi_i)}^2 \eta_i (y_i^{(t-1)} - f(\phi_i)) \end{aligned}$$

则损失函数变为，与源中的特定目标（变量）

$$\sum_{i=1}^n [g_i / f(\phi_i) + \frac{1}{2} h_i f_{\phi_i}^2(\phi_i) + \Omega(f_t)]$$

这成为我们求解的优化目标。这定义了一个非凸的优化目标函数的近似取决于 g_i 和 h_i ，这就是xgboost的自定义损失函数的方式。我们可以使用完整的 η_i 来求解每个损失函数，包括正则，并把它替换为 g_i 和 h_i 值！

对比可以看到，其它损失函数泰勒展开之后去掉常数最终的形式和mse的不泰勒展开的形式是完全一致的（mse的二阶梯为常数1，一阶梯度是 $y_{\text{pred}} - y_{\text{True}}$ ），这么做的好处是，这样的话，1、 xgboost在对mse的损失函数设计完求解器之后，这一套代码可以直接复用给别的损失函数来使用，因为我们如果不做二阶泰勒展开的话，比如新的损失函数是二元交叉熵，在工程设计上，我们还要将损失函数的求导，然后把求导之后的式子写出来：

$$\frac{(\log(\phi^{(i)}) - y^{(i)})e^{(i)}}{1 + e^{(i)}} \quad \text{二元交叉熵的一阶梯度}$$

设计一个新的求解器去求解，很麻烦。

而进行了这样的设计之后，后续如果还有一些别的损失函数，底层的求解mse的代码可以直接使用，使用者只需要自行去求解新的损失函数的一阶梯度和二阶梯度的表达式，然后通过xgboost的自定义损失函数的功能就可以实现使用完备的xgboost的框架来求解自己的损失函数的最优值了。

2、关于速度的问题，gbdt的前向分布的求解思路可以说就和我们常见的逻辑回归求解的梯度下降是类似的，线性回归的梯度下降每一轮通过更新参数的方式接近损失函数的最优值，而gbdt则是用基学习器去拟合，相对而言，xgboost类似于使用牛顿法来求解线性回归，所以下面从牛顿和梯度下降的角度来阐述，的实际上我们常说的牛顿法比梯度下降法快是不准确的，应该是牛顿法的收敛速度要比梯度下降法快，也就是说牛顿法使用的迭代次数相对于梯度下降法要更少，但是由于涉及到计算二阶导的信息，牛顿法不一定在算法训练的时间上总比梯度下降法快，只是相对于梯度下降法而言，更少的迭代达到最优，这一点来看，并不算是优势。

- xgboost在代价函数里加入了正则项，用于控制模型的复杂度。正则项里包含了树的叶子节点个数、每个叶子节点上输出的score的L2模的平方和。从Bias-variance tradeoff角度来讲，正则项降低了模型的variance，使学习出来的模型更加简单，防止过拟合，这也是xgboost优于传统GBDT的一个特性。

- Shrinkage (缩减)，相当于学习速率 (xgboost中的eta)。xgboost在进行完一次迭代后，会将叶子节点的权重乘上该系数，主要是为了削弱每棵树的影响，让后面有更大的学习空间。实际应用中，一般把eta设置得小一点，然后迭代次数设置得大一点。（补充：传统GBDT的实现也有学习速率）

- 列抽样 (column subsampling)。xgboost借鉴了随机森林的做法，支持列抽样，不仅能降低过拟合，还能减少计算，这也是xgboost异于传统gbdt的一个特性。

- 对缺失值的处理。对于特征的值有缺失的样本，xgboost可以自动学习出它的分裂方向。xgboost的缺失值处理也是比较一个大的概念，实际上在关于cart tree的缺失值处理之前是有很多方案的，例如插补法、cart的代理分裂功能，但是代理分裂功能在构建大量tree的时候是非常耗时的，并且实际上在gbm的框架下单棵树的性能实际上并不是那么重要，xgboost的处理方式很简单，将缺失样本分别放入左右两枝比较最终的gain的大小，xgboost的所谓的稀疏感知，可以看到实际上就是对缺失值的处理，现在没有缺失的值上分裂，然后把缺失值分别带入左节点算一下分裂后的增益，再带入右节点算一下分裂后的增益然后去其中大的一个作为最终的分裂方案。如果训练中没有数据缺失，预测时出现了数据缺失，则默认被分类到右节点

- ,

- xgboost工具支持并行。boosting不是一种串行的结构吗?怎么并行的? 注意xgboost的并行不是tree粒度的并行，xgboost也是一次迭代完才能进行下一次迭代的（第t次迭代的代价函数里包含了前面t-1次迭代的预测值）。xgboost的并行是在特征粒度上的。我们知道，决策树的学习最耗时的一个步骤就是对特征的值进行排序（因为要确定最佳分割点），xgboost在训练之前，预先对数据进行了排序，然后保存为block结构，后面的迭代中重复地使用这个结构，大大减小计算量。这个block结构也使得并行成为了可能，在进行节点的分裂时，需要计算每个特征的增益，最终选增益最大的那个特征去做分裂，那么各个特征的增益计算就可以开多线程进行。但是预排序的问题就在于我们实际上要消耗很多的内存去存放排序完毕之后的原始数据；

- 可并行的近似直方图算法。树节点在进行分裂时，我们需要计算每个特征的每个分割点对应的增益，即用贪心法枚举所有可能的分割点。当数据无法一次载入内存或者在分布式情况下，贪心算法效率就会变得很低，所以xgboost还提出了一些近似算法，包括：

1.4 xgboost的缺点

- 1、难以处理高维稀疏的数据，因为tree本身的正则化对于高维稀疏的数据情况不像l1正则化能够带来有效的约束，并且tree本身的分裂在稀疏的情况下显著性很差；
- 2、对于异常点（注意是异常点不是异常值）较为敏感，因为gbdt会在因为异常导致预测误差特别大地样本上不断地去用新的tree来拟合，导致模型太过拟合异常样本，最终的结果就是泛化性能差；例如在回归问题中，假设标签是【1.6, 2.6, 3.5, 1.5, 1000】，这种情况下gbdt会不断的用新tree去拟合标签为1000的样本的负梯度。
- 3、集成模型本身的计算复杂度都是比较高的，训练耗时；
- 4、xgBoosting采用level-wise生成决策树，同时分裂同一层的叶子，从而进行多线程优化，不容易过拟合，但很多叶子节点的分裂增益较低，没必要进行跟进一步的分裂，这就带来了不必要的开销，不过目前新版的xgboost已经引入了level-wise和leaf-wise两种生长模式了，这两种生长模式没有绝对的好坏之分；

1.1 lightgbm简介

Lightgbm，微软出品，解决xgboost训练速度慢和内存占用大的问题，当然实际上还实现了很多新的特性，不过整体的gbdt的框架是差不多的

训练时间的对比：

Data	xgboost	xgboost_hist	LightGBM	Data	xgboost	xgboost_hist	LightGBM
Higgs	4.051GB	3.791GB	0.868GB	Higgs	3794.34 s	551.900 s	238.595513 s
Yahoo LTR	1.907GB	1.458GB	0.831GB	Yahoo LTR	674.122 s	265.902 s	150.18544 s
MS LTR	5.461GB	3.654GB	0.886GB	MS LTR	1251.27 s	345.201 s	215.320316 s
Expo	1.551GB	1.093GB	0.543GB	Expo	1607.35 s	588.253 s	138.504179 s
Allstate	6.237GB	4.890GB	1.027GB	Allstate	2867.22 s	1355.71 s	343.08447 s

Lightgbm继承了xgboost的大部分优点，并且在此之上发扬光大，lightgbm在xgboost上的改进包括：

- 1.直方图优化，对连续特征进行分桶，在损失了一定精度的情况下大大提升了运行速度，并且在gbm的框架下，基学习器的“不精确”分箱反而增强了整体的泛化性能；
- 2、直接支持对于类别特征进行训练（实际上内部是对类别特征做了梯度编码的操作了），对类别特征的分割做了改进

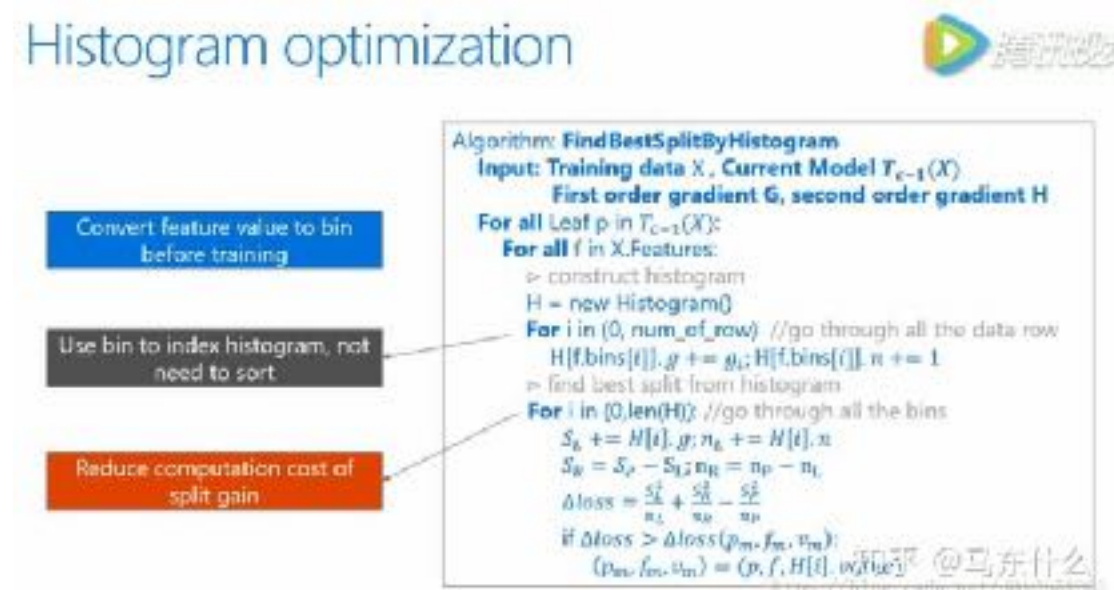
第一个比较：

xgboost使用了pre-sorted来对特征进行预排序，一方面内存的消耗比较大，因为需要单独开辟新的空间来存放排序后的特征矩阵，并且因为不同特征的排列顺序一般是不同的，所以我们仍旧需要一一列特征的去计算最佳的分裂节点并且比较所有特征列的最佳分裂节点之后得到最优的分裂特征与对应的分裂节点，如此一来，对于单个特征，原始的exact（我们这里暂时先不讨论xgboost的各种近似等改良之后的分裂方案）需要遍历data（去重之后样本个数-1）次，则features一共要遍历data*features次。

相比较而言，lightgbm使用了histogram，直方图算法，实际上简单来说就是对所有特征进行了统一的分箱处理，比如有连续特征A有10000个不同的取值，进行分箱之后将10000个值划分到5个不同的箱子中去，这样的话我们每次就只要遍历4个间隔就可以得到最优分裂点了。

https://blog.csdn.net/anshuai_aw1/article/details/83040541blog.csdn.net

看了一下源码，lightgbm的分箱还是比较细致而复杂的。



lightgbm详细提到直方图算法的资料，我一开始看的时候还觉得疑惑，难道每次分裂的时候都要重新创建直方图？？后来发现理解错了，实际上，我们需要注意一个细节，**xgboost**初始化的值默认为**0.5**，这个，各位可以自己自定义损失函数然后输出每一轮的预测值就会发现无论是什么损失函数**第一轮的预测都是默认全0.5**，而**lightgbm**则是默认输出全为**0**的。所以在分箱（或者叫分桶。。whatever，都行）之前是先进行这么一步，第一颗树就是一条简单的规则，输出常数，然后根据第一棵树的预测计算一阶梯度和二阶梯度，并且分箱。接着再进行下一轮的拟合，**xgboost**相对来说就是少了分箱（分桶）这一步。

关于分桶，lightgbm针对连续特征和类别特征的分桶策略是不同的，先来看看连续特征的分桶策略：

```
std::vector<double> GreedyFindBin(const double* distinct_values, const int* counts,
                                int num_distinct_values, int max_bin, size_t total_cnt, int min_data_in_bin) {
    // counts为特征取值计数，max_bin最大分桶数量这个参数可以由用户自己定义，默认值是255，min_data_in_bin//这个参数也可以由用户自定义，默认是3，这些都在lightgbm的IO Parameters 里面作为超参数存在
```

```

std::vector<double> bin_upper_bound;

CHECK(max_bin > 0);

if (num_distinct_values <= max_bin) { //如果独立的特征值的数量比max_bin还小,
bin_upper_bound.clear();

    int cur_cnt_inbin = 0;

    for (int i = 0; i < num_distinct_values - 1; ++i) {

        cur_cnt_inbin += counts[i]; //counts[i]表示的是每一个独立的特征值的计数
        if (cur_cnt_inbin >= min_data_in_bin) {

            auto val = Common::GetDoubleUpperBound((distinct_values[i] +
distinct_values[i + 1]) / 2.0); //如果某个特征值的取值数大于min_data_in_bin, 则得到
一个切分点, 切分点为对应的相邻的两个distinct_values的平均数
            if
(bin_upper_bound.empty() || !
Common::CheckDoubleEqualOrdered(bin_upper_bound.back(), val)) {

                bin_upper_bound.push_back(val); //将切分点存储起来, 进入下一轮循环
                cur_cnt_inbin = 0;

            }

        }

    }

    cur_cnt_inbin += counts[num_distinct_values - 1];

    bin_upper_bound.push_back(std::numeric_limits<double>::infinity()); //
上限值是正无穷大
}

else { // 特征取值比max_bin来得大, 说明几个特征取值要共用一个bin
    if
(min_data_in_bin > 0) { //判断max_bin的大小
        max_bin = std::min(max_bin,
static_cast<int>(total_cnt / min_data_in_bin));

        max_bin = std::max(max_bin, 1);

    }

    double mean_bin_size = static_cast<double>(total_cnt) / max_bin;

    // mean size for one bin
    int rest_bin_cnt = max_bin;

    int rest_sample_cnt = static_cast<int>(total_cnt);

    std::vector<bool> is_big_count_value(num_distinct_values, false);

    // 标记一个特征取值数超过mean, 因为这些特征需要单独一个bin
    for (int i
= 0; i < num_distinct_values; ++i) {

```

```

        if (counts[i] >= mean_bin_size) {

            is_big_count_value[i] = true;

            --rest_bin_cnt;

            rest_sample_cnt -= counts[i];

        }

    }

    //剩下的特征取值中平均每个bin的取值个数    mean_bin_size =
static_cast<double>(rest_sample_cnt) / rest_bin_cnt;

    std::vector<double> upper_bounds(max_bin,
std::numeric_limits<double>::infinity());

    std::vector<double> lower_bounds(max_bin,
std::numeric_limits<double>::infinity());

    int bin_cnt = 0;

    lower_bounds[bin_cnt] = distinct_values[0];

    int cur_cnt_inbin = 0;

    for (int i = 0; i < num_distinct_values - 1; ++i) {

        if (!is_big_count_value[i]) {

            rest_sample_cnt -= counts[i];

        }

        cur_cnt_inbin += counts[i];

        // need a new bin, 当前的特征如果是需要单独成一个bin, 或者当前几个特征计数超过
了mean_bin_size, 或者下一个是需要独立成列的    if (is_big_count_value[i] ||
cur_cnt_inbin >= mean_bin_size ||

        (is_big_count_value[i + 1] && cur_cnt_inbin >= std::max(1.0,
mean_bin_size * 0.5f))) {

            upper_bounds[bin_cnt] = distinct_values[i]; // 第i个bin的最大就是
distinct_values[i]了    ++bin_cnt;

            lower_bounds[bin_cnt] = distinct_values[i + 1]; //下一个bin的最小就是
distinct_values[i + 1], 注意先++bin了    if (bin_cnt >= max_bin - 1)
{ break; }

            cur_cnt_inbin = 0;

            if (!is_big_count_value[i]) {

                --rest_bin_cnt;

                mean_bin_size = rest_sample_cnt /
static_cast<double>(rest_bin_cnt);

```

```

    }

    }

}

++bin_cnt;

// update bin upper bound      bin_upper_bound.clear();

for (int i = 0; i < bin_cnt - 1; ++i) {

    auto val = Common::GetDoubleUpperBound((upper_bounds[i] +
lower_bounds[i + 1]) / 2.0);

    if (bin_upper_bound.empty() || !
Common::CheckDoubleEqualOrdered(bin_upper_bound.back(), val)) {

        bin_upper_bound.push_back(val);

    }

}

// last bin upper bound
bin_upper_bound.push_back(std::numeric_limits<double>::infinity());

}

return bin_upper_bound;

}

```

开始之前，假设我们要处理特征A，首先对A进行排序然后有类似于np.unique的操作，返回A的所有不重复取值和每一个取值对应出现的次数，比如[1,2,3,4,5,6,1,2,3,4]处理完毕之后就得到了distinct_values=[1,2,3,4,5,6],counts=[2,2,2,2,1,1],对于连续值的分桶，要分情况讨论：

为了便于描述这里我们以一个假设的数字序列为例子，【1，1，1，1，2，2，3，3，3，3，4，4，4，4，5，5，5，6】，首先进行排序，（这里已经排序完毕），然后计算得到：distinct_values=[1,2,3,4,5,6],counts=[4,2,4,4,3,1]，这个时候开始分桶

1、假设max_bin=10,此时distinct_values的值得比max_bin小，那么说明一个bin可以存放一个值，那么我们就开始从小到大开始进行分桶，首先1分为第一个桶，我们命名为“1桶”，这个时候分桶的切分点不是1，而是1和下一个数字2的平均数即 $(1+2)/2=1.5$ ，接着到2，此时这里有一个小细节需要注意，lightgbm有一个不起眼的参数参数min_data_in_bin默认值为3，也就是如果连续值中的某一个数的取值小于3是不能单独分桶的，这是一个可以调节的超参数，想象一下，当min_data_in_bin=1，而连续值没有一个重复的，那么最终分

桶的结果和没分桶一样，从这个角度来看，这里实际上起到了一定正则化的作用，把取值差不多的但是出现次数少的数字合并到一块儿，避免了太过精细的切分。那么2无法合并，就要和后面的3合并成一个桶，则【2, 2, 3, 3, 3, 3】合并成一个桶，我们称之为“2桶”，此时需要注意，因为在2的时候由于2的数量不够无法单独分桶，所以切分点不是 $(2+3)/2=2.5$ 而应该与下一个连续值4相关联，切分点应该是 $(3+4)/2=3.5$ ，下一个连续值是4，单独为1个桶，切分点为 $(4+5)/2=4.5$ ，接着5数量为3 可以单独分桶不需要和6合并，因此切分点为 $(5+6)/2=5.5$ ，注意到这个时候整个序列已经分桶完毕，6是最大值，所以此时最终的上限为infinite即无穷大，另外分桶的初始端也需要增加一个-infinite，即负无穷大，这个时候我们就得到了所有的切分点了，即【负无穷大，1.5, 3.5, 4.5, 5.5,正无穷大】，根据这个分桶规则进行相应的映射就可以了，和常规的分箱操作很类似，只不过这里是有序的分桶而不是无序的分桶，那么我们将原始的数据映射一下可以得到【1, 1, 1, 1, 2, 2, 2, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 5】之后如果需要针对这个特征进行切分，我们只需要在1.5,3.5,4.5, 5.5进行切分就可以了，然后再去进行后续的计算，这样的操作还有一个很大的好处就是我们不需要保存原始的排序之后的特征，仅仅需要保存这些int类型的分桶结果就可以，这样实际上相对于float类型数据的存储来说是要大大减少占用量的。

（这里后续仔细看了下实际上lightgbm是按照从大到小的顺序进行分桶的，不过没关系不影响理解）

2、当然，一般情况下，lightgbm的max_bin是250（差不多这值，懒得查官方文档了），而连续值的不重复取值往往要大于这个max_bin的，所以就进入第二种情况了。

首先，在lightgbm会定义一个真正的bin的数量，而不是单纯的直接使用max_bin，根据源代码可以知道，假设有10000个样本，也就是特征有10000个数，假设min data in bin=3，此时会用 $10000/3=3333.33$ ，然后比较max bin和这个数值的大小，取其中最小的数，不过一般数据量大一点基本都是取到max bin的。。。。，然后会计算一个叫 mean_bin_size 的常数，以10000个样本为例子， $\text{mean_bin_size}=10000/250=40$ ，这个mean_bin_size是用来界定后面的“大数”和“小数”的，如果distinct values的count计数大于mean_bin_size，则这个数字会被标记为“大数”，其余的数字就是“小数”了，此时暂时是还没有进行分桶操作的需要注意。

这里还有一个准备工作的细节需要注意，假设我们有10000个样本，在上面的操作之后得到了5个“大数”并且5个“大数”一共占了5000个样本，此时我们的mean bin size 要重新计算， $(10000-5000)/(250-5)$ ，就是把“大数”和对应占据的桶数去掉之后重新计算一次mean bin size。

上述准备工作完毕，接下来就正式进入分桶的操作了，还是一样排序之后，从 distinct values 的大到小的顺序开始分桶，首先进行一个判断：

1、当前的 distinct value 是否在之前的处理中被标记为“大数”，如果被标记为“大数”则此时同时可以得到两个切分点，上限就是当前的这个被定义为“大数”的 distinct value，下限就是下一个较小的 distinct value。和情况一的切分点的确定很不相同，这里需要注意一下。比如当前 distinct 为“大数”，其值为 100，下一个 distinct value（不管它是大数还是小数）比如是 90，此时我们就得到了两个切分点 100 和 90，和情况 1 截然不同注意区分。

2、如果当前的 distinct value 并没有被标记为“大数”而是一个“小数”，则和情况 1 类似，我们进行一个累计的操作，不过这里有很多细节的地方需要注意，当我们进行累计的时候，以上面的 10000 样本的例子为例，分完“大数”之后小数样本一共 5000 个，则假设当前的 distinct value 一共有 10 个，则 $5000 - 10$ ，同时 max bin 在分完大数之后是 $250 - 5 = 245$ ，此时经过累计之后要减去 1，也就是 244，然后我们重新再次计算一下 $\text{mean bin size} = 5000 / 244$ 进入下一轮的循环中去。

这个时候就又有两种情况了：

(1)、我们上面进行了累计的时候，假如很不巧，第 t 轮的累计的数量没有超过第 $(t-1)$ 轮重新计算出来的 mean bin size，并且第 $(t+1)$ 轮遇到了一个“大数”，那很尴尬，“大数”需要单独分桶，但是此时我们又没累计够，这个时候我们没有办法，退而求其次，如果累计值大于等于 $\max(1, \text{mean bin size} * 0.5)$ 我们也勉强把这些累计的数当作“大数”，此时得到切分点的过程也是一样，当前的 distinct value 和下一个较小的 distinct value 得到两个切分点。

(2)、我们上面进行累计的时候，运气好，在遇到大数之前，第 t 轮的累计的数量超过了第 $(t-1)$ 轮重新计算的 mean bin size，则得到切分点的过程一样，当前的 distinct value 和下一个较小的 distinct value 得到两个切分点。

这样，针对情况 2 我们就得到了所有的切分点，然后上下限加入正无穷大和负无穷大，就得到了最终的分桶的所有的切分点和切分区间了。

直方图的优缺点：

直方图算法的基本思想是将连续的特征离散化为 k 个离散特征，同时构造一个宽度为 k 的直方图用于统计信息（含有 k 个 bin）。利用直方

图算法我们无需遍历数据，只需要遍历 k 个 bin 即可找到最佳分裂点。

我们知道特征离散化的具有很多优点，如存储方便、运算更快、鲁棒性强、模型更加稳定等等。对于直方图算法来说最直接的有以下两个优点（以 $k=256$ 为例）：

- **内存占用更小**：XGBoost 需要用 32 位的浮点数去存储特征值，并用 32 位的整形去存储索引，而 LightGBM 只需要用 8 位去存储直方图，相当于减少了 $1/8$ ；
- **计算代价更小**：计算特征分裂增益时，XGBoost 需要遍历一次数据找到最佳分裂点，而 LightGBM 只需要遍历一次 k 次，直接将时间复杂度从 $O(n)$ 降低到 $O(k)$ ，而我们知道 $k \ll n$ 。 <https://zhuanlan.zhihu.com/p/87885678>

虽然将特征离散化后无法找到精确的分割点，可能会对模型的精度产生一定的影响，但较粗的分割也起到了正则化的效果，一定程度上降低了模型的方差，并且在gbm的框架下实际上单棵tree的精度并不是非常关键，毕竟质量不够，数量（更多的tree）来凑。

2) 直方图加速

在构建叶节点的直方图时，我们还可以通过父节点的直方图与相邻叶节点的直方图相减的方式构建，从而减少了一半的计算量。在实际操作过程中，我们还可以先计算直方图小的叶子节点，然后利用直方图作差来获得直方图大的叶子节点。



类别特征的处理相对来说要简单的多。

参考：（还解释了为什么常规的gbdt和xgboost不适用于类别特别多的特征）

关于sklearn中的决策树是否应该用one-hot编码https://blog.csdn.net/anshuai_aw1/article/details/83275299
blog.csdn.net

```
// sort by counts 特征取值按出现的次数排序（大到小）
Common::SortForPair<int, int>(counts_int, distinct_values_int, 0, true);

    // avoid first bin is zero    if (distinct_values_int[0] == 0) {

        if (counts_int.size() == 1) {

            counts_int.push_back(0);

            distinct_values_int.push_back(distinct_values_int[0] + 1);

        }

        // 交换counts_int[0]和counts_int[1]的值
std::swap(counts_int[0], counts_int[1]);

        std::swap(distinct_values_int[0], distinct_values_int[1]);

    }

    // will ignore the categorical of small counts    int cut_cnt =
static_cast<int>((total_sample_cnt - na_cnt) * 0.99f);

    size_t cur_cat = 0;

    // categorical_2_bin_ (unordered_map类型) 将特征取值到哪个bin和——对应起
来
    categorical_2_bin_.clear();

    // bin_2_categorical_ (vector类型) 记录bin对应的特征取值
bin_2_categorical_.clear();

    int used_cnt = 0;

    max_bin = std::min(static_cast<int>(distinct_values_int.size()),
max_bin);

    cnt_in_bin.clear();

    // 类别特征值已经按数量从大到小排列，累积特征值的数目，放弃后1%的类别特征值，即忽
略一些出现次数很少的特征取值    while (cur_cat < distinct_values_int.size()

        && (used_cnt < cut_cnt || num_bin_ < max_bin)) {

            if (counts_int[cur_cat] < min_data_in_bin && cur_cat > 1) {

                break;

            }

            //为bin_2_categorical_和categorical_2_bin_赋值
bin_2_categorical_.push_back(distinct_values_int[cur_cat]);

            categorical_2_bin_[distinct_values_int[cur_cat]] =
static_cast<unsigned int>(num_bin_);
```

```

        used_cnt += counts_int[cur_cat];

        cnt_in_bin.push_back(counts_int[cur_cat]);

        ++num_bin_;

        ++cur_cat;
    }

```

首先lightgbm会去计算每一个类别的counts以及所有类别的distinct value。然后忽略后1%的类别特征：

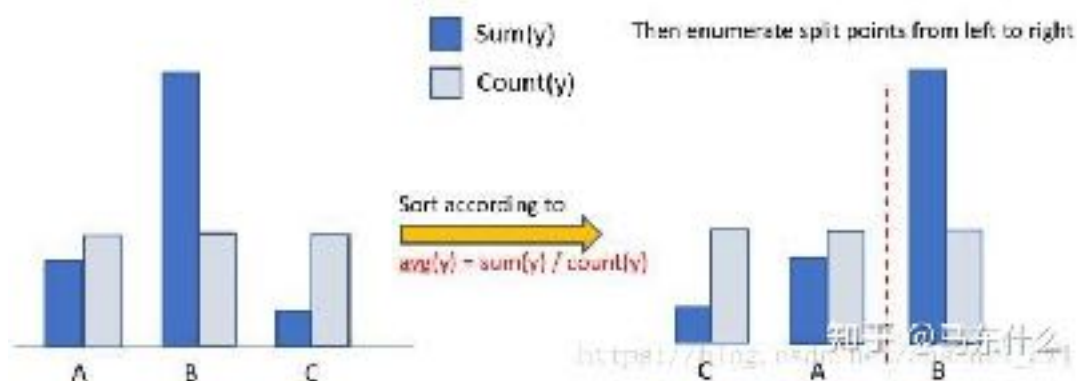
`int cut_cnt = static_cast<int>((total_sample_cnt - na_cnt)*0.99f);`，如果类别特征出现的次数很少会被直接忽略掉。。。，所以最终是99%的类别特征进行分桶，因此为了不损失信息。。。最好人工事先手动对类别做一些合并免得最后都被忽略掉了

另外需要注意的是，

- 每个分类组的数据量最少
 - `max_cat_threshold` ∞ , default = 32, type = int, 约束: `max_cat_threshold > 0`
 - 用于分类特征
 - 限制分类特征中的最大取值点
- `cat_l2` ∞ , default = 10.0, type = double, 约束: `cat_l2 >= 0.0`
 - 用于分类特征
 - 类别拆分中的L2正则化
- `cat_smooth` ∞ , default = 10.0, type = double, 约束: `cat_smooth >= 0.0`
 - 用于分类特征
 - 这样可以减少噪声对分类特征的影响，尤其是对于数据较少的分类
- `max_cat_to_onehot` ∞ , default = 4, type = int, 约束: `max_cat_to_onehot > 0`
 - 当一个特征的类别数小于或等于时 `max_cat_to_onehot`，将使用“列多拆分算法”

当类别的数量小于`max_cat_to_onehot`的时候，lightgbm是不分桶直接按照ovr的方式处理的，比如有类别A B C，则按照A, BC; B, AC; C, AB来进行分裂增益的分别计算。

如果类别数量大于`max_cat_to_onehot`则按照lightgbm自己独特的方式来进行分裂，首先：



官方这里给的是回归的案例，mse的二阶梯度为1所以正好二阶梯度的和就是count(y)，注意到上图的参数里有一个cat_smooth的参数用于正则化，降低了数量少的类别特征的噪声影响，所以实际上每一个类别被分桶之后首先计算的是这个桶中所有样本的：

一节梯度之和/(二阶梯度之和+正则化系数cat_smooth)

得到这个值之后按照这个值将所有的类别的桶从大到小排序，

然后注意到还有max_cat_threshold这个参数，lightgbm默认是从左到右和从右往左分别搜索max_cat_threshold个桶然后计算其中最大的增益切分点的（真是有够粗糙的切分。。。）

这里的cat_l2正则化，不太明白是用在哪里的希望有大佬告知。目前猜测应该是切分计算增益的时候分母项增加了一项针对切分完毕之后两边的所有桶的一节梯度之和/(二阶梯度之和+正则化系数cat_smooth)之平方和。

2.goss 树的引入；

Lightgbm实际上实现了dart和goss两种生长模式，lightgbm的dart和xgboost中的dart在算法上略有不同，这里先介绍下goss吧：

单边梯度采样算法。

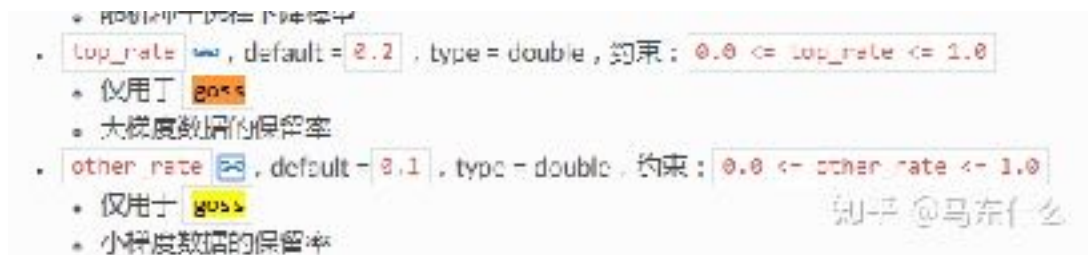
Algorithm 2: Gradient-based One-Side Sampling

Input: I : training data, d : iterations
Input: a : sampling ratio of large gradient data
Input: b : sampling ratio of small gradient data
Input: $loss$: loss function, L : weak learner
 $models \leftarrow \{\}$, $fact \leftarrow \frac{1-a}{b}$
 $topN \leftarrow a \times \text{len}(I)$, $randN \leftarrow b \times \text{len}(I)$
for $i = 1$ to d do
 $preds \leftarrow models.predict(I)$
 $g \leftarrow loss(I, preds)$, $w \leftarrow \{1, 1, \dots\}$
 $sorted \leftarrow \text{GetSortedIndices}(\text{abs}(g))$
 $topSet \leftarrow sorted[1:topN]$
 $randSet \leftarrow \text{RandomPick}(sorted[topN:\text{len}(I)], randN)$
 $usedSet \leftarrow topSet + randSet$
 $w[randSet] \times = fact$ \triangleright Assign weight $fact$ to the small gradient data.
 $newModel \leftarrow L(I[usedSet], -g[usedSet], w[usedSet])$
 $models.append(newModel)$

知乎 @马东什么

goss的实现思路和focal loss比较相似，focal loss是对梯度小的样本进行降权处理，而goss是直接进行一个部分丢弃的设计，目的都是为了在迭代中把梯度小的样本的影响降，因为梯度小的样本本身已经拟合的很充分了，继续迭代只会浪费时间，具体思路是：

- 1、每一轮迭代之后我们都能得到每一个样本的一阶梯度 g ，根据 g 的大小对所有样本进行排序；
- 2、



根据top_rate，我们取前20%的大梯度样本进入下一轮迭代，同时根据other_rate从剩下的80%的小梯度样本中随机采样10%的样本也进入下一轮的迭代，这样实际上每一轮类似于做了一个30%的bagging的采样，不过需要注意的是使用goss的时候我们是无法设置sub_sample这个参数的，因为如果再进行bagging然后再使用goss很可能导致bagging出来的样本都是小梯度样本，这样是我们就要用新树来拟合小梯度的样本，这是没有意义的。

在比赛中，goss的精度基本总要比gbdt差一点点，不过在实际工作中可以使用这样的方法，很多时候工作中对较小的精度的差距并不敏感，而goss在精度大致相当的情况下可以非常迅速的训练完毕。

另外需要注意的一个细节就是：

$$\tilde{V}_j(d) = \frac{1}{n} \left(\frac{(\sum_{x_i \in A_l} g_i + \frac{1-a}{b} \sum_{x_i \in B_l} g_i)^2}{n_l^j(d)} + \frac{(\sum_{x_i \in A_r} g_i + \frac{1-a}{b} \sum_{x_i \in B_r} g_i)^2}{n_r^j(d)} \right)$$

小梯度样本在计算的时候goss内部是对这些小梯度样本进行加权了，比如我们大梯度样本的采样率是20%，即a=20%，而小梯度样本的采样率为10%，即b=10%，则加权的结果为(1-20%)/10%=8，小梯度样本权重变成原来的8倍了，这么做是为了尽量保证特征的分布不发生太过的变化。

3.ebf，对稀疏特征做了“捆绑”的优化功能；

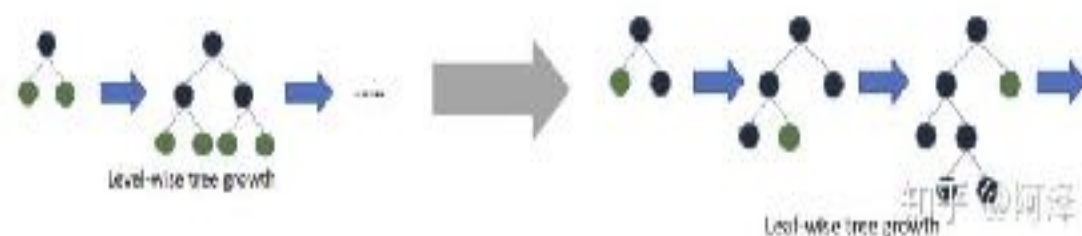
- `enable_bundle` \Rightarrow 默认 `true` , 类型 `bool` , 别名: `is_enable_bundle` , `bundle`
 - 将此选项设置 `false` 为禁用该功能(即 FFR) , 这在 LightGBM 高效稀疏提升决策树中进行了描述
 - 注意: 禁用此选项可能会导致稀疏数据训练的收敛速度变慢
- `max_conflict_rate` \Rightarrow , default = 0.0 , type = double , 约束: $0.0 \leq \text{max_conflict_rate} < 1.0$
 - 树中内部节点的最大冲突率
 - 将此设置 0.0 为禁止冲突并提供准确的结果
 - 将此值设置为更大的值可实现更快的速度
- `is_enable_sparse` \Rightarrow , 默认 `true` , 类型=布尔, 别名: `is_sparse` , `enable_sparse` , `sparse`
 - 用于启用/禁用稀疏优化
- `sparse_threshold` \Rightarrow , default = 0.5 , type = double , 约束: $0.0 \leq \text{sparse_threshold} < 1.0$
 - 将观察视为稀疏观察的阈值百分比
- `use_missing` \Rightarrow , 默认 `true` , 类型= bool
 - 设置 `false` 为禁用缺失值的特殊处理
- `zero as missing` \Rightarrow , 默认 `false` , 类型= bool
 - 设置 `true` 为将所有未知或缺失值 (包括 `bsvm / sparse` 矩阵中未显示的项)
 - 将此设置 `false` 为 `nan` 作为缺失值

4. 树的生长方式由 level-wise 变成 leaf-wise;

在建树的过程中有两种策略:

- Level-wise: 基于层进行生长, 直到达到停止条件;
- Leaf-wise: 每次分裂增益最大的叶子节点, 直到达到停止条件。

XGBoost 采用 Level-wise 的增长策略, 方便并行计算每一层的分裂节点, 提高了训练速度, 但同时也因为节点增益过小增加了很多不必要的分裂, 降低了计算量; LightGBM 采用 Leaf-wise 的增长策略减少了计算量, 配合最大深度的限制防止过拟合, 由于每次都需要计算增益最大的节点, 所以无法并行分裂。



这两种生长方式没有绝对的好坏;

1. 工程上:

2.2.1 特征并行

传统的特征并行算法在于对数据进行垂直划分，然后使用不同机器找到不同特征的最优分裂点，基于通信整合得到最佳划分点，然后基于通信告知其他机器划分结果。

传统的特征并行方法有个很大的缺点：需要告知每台机器最终划分结果，增加了额外的复杂度（因为对数据进行垂直划分，每台机器所含数据不同，划分结果需要通过通信告知）。

LightGBM 则不进行数据垂直划分，每台机器都有训练集完整数据，在得到最佳划分方案后可在本地执行划分而减少了不必要的通信。

2.2.2 数据并行

传统的数据并行策略主要为水平划分数据，然后本地构建直方图并整合成全局直方图，最后在全局直方图中找出最佳划分点。

这种数据划分有一个很大的缺点：通讯开销过大。如果使用点对点通信，一台机器的通讯开销大约为 $\frac{1}{n}$ ；如果使用集成的通信，则通讯开销为 $\frac{1}{n}$ ，

LightGBM 采用分散规约（Reduce scatter）的方式将直方图整合的任务分摊到不同机器上，从而降低通信代价，并通过直方图做差进一步降低不同机器间的通信。

2.2.3 投票并行

针对数据量特别大特征也特别多的情况下，可以采用投票并行。投票并行主要针对数据并行时数据合并的通信代价比较大的瓶颈进行优化，其通过投票的方式只合并部分特征的直方图从而达到降低通信量的目的。

大致步骤为两步：

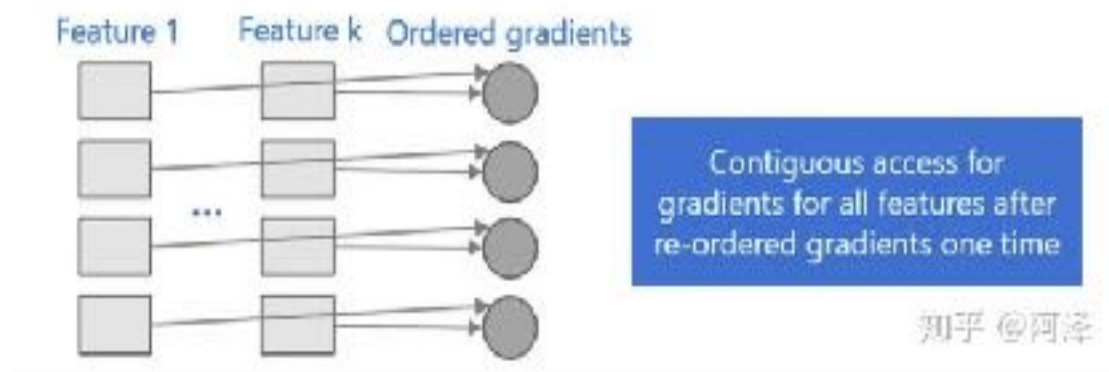
1. 本地找出 Top K 特征，并基于投票筛选出可能是最优分割点的特征；
2. 合并时只合并每个机器选出来的特征。

2.2.4 缓存优化

上边说到 XGBoost 的预排序后的特征是通过索引给出的样本梯度的统计值，因其索引访问的结果并不连续，XGBoost 提出缓存访问优化算法进行改进。

而 LightGBM 所使用直方图算法对 Cache 天生友好：

1. 首先，所有的特征都采用相同的方法获得梯度（区别于不同特征通过不同的索引获得梯度），只需要对梯度进行排序并可实现连续访问，大大提高了缓存命中；
2. 其次，因为不需要存储特征到样本的索引，降低了存储消耗，而且也不存在 Cache Miss 的问题。



Catboost:

原理：

CatBoost是俄罗斯的搜索巨头Yandex在2017年开源的机器学习库，和lightgbm、xgboost并成为gbdt三大主流神器库，它是一种基于对称决策树（oblivious trees）算法的参数少、支持类别型变量和高准确性的GBDT框架，主要解决的痛点是高效合理地处理类别型特征，另外提出了新的方法来处理

梯度偏差 (Gradient bias) 以及预测偏移 (Prediction shift) 问题，提高算法的准确性和泛化能力。

整体上来说，个人感觉catboost的设计灵活度确实很高，但是灵活度高的同时也给人一种很杂的感觉。。。因为在很多的小细节上做了改进，让人有一种很不想深入研究的欲望。。。

目标函数：

<https://catboost.ai/docs/concepts/algorithm-score-functions.html>catboost.ai

首先我们要搞清楚xgboost中损失函数与打分函数的关系：

xgboost的损失函数优化推导之后得到的打分函数，

$$\begin{aligned}
 L^{(t)} &= \sum_{i=1}^n [g_i f_t(x_i) - \frac{1}{2} h_i f_t^2(x_i)] - \Omega(f_t) \\
 &= \sum_{i=1}^n [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Upsilon T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \\
 &= \sum_{j=1}^T [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T \quad (\text{式9})
 \end{aligned}$$

对式9进行求导：

$$\begin{aligned}
 \text{令 } \frac{\partial L^{(t)}}{\partial w_j} &= 0 \\
 \Rightarrow (\sum_{i \in I_j} g_i) - (\sum_{i \in I_j} h_i + \lambda) w_j &= 0 \\
 \Rightarrow (\sum_{i \in I_j} h_i + \lambda) w_j &= - \sum_{i \in I_j} g_i \\
 \Rightarrow w_j^* &= - \frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda} \quad (\text{式10})
 \end{aligned}$$

这里推导得到了叶子节点的权重公式，

将 (式10) 代入 (式9) 中得到 (式11) :

$$L^{(t)} = -\frac{1}{2} \sum_{j=1}^T \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \Upsilon T \quad (式11)$$

令 $G_i = \sum_{i \in I_j} g_i, H_i = \sum_{i \in I_j} h_i$ 则, (式11) 可以简化为 (式12)

$$L^{(t)} = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \Upsilon T \quad (式12)$$

知乎 @马东什么

权重公式代入回损失函数最终得到了打分函数, 而我们需要通过打分函数来决定基树的分裂:

$$Gain = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} - \frac{G_H^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma \quad (式13)$$

知乎 @马东什么

打分函数是xgboost对损失函数的处理在树的分裂过程中的表现, 这也是后xgboost时代, gbd算法的一个比较大的改变, 基树不再是单纯的cart树使用gini进行分裂而是使用上面的梯度信息进行分裂的指导。

而catboost的损失函数实际上既实现了最原始的gbdt的一阶梯度的版本也实现了xgboost的泰勒展开引入一二阶梯度的版本, 这一点在其提供的score function中可见一斑:

评分功能	描述
L2	在计算过程中使用一阶导数。
余弦 (不能与 Lossguide 树生长策略一起使用)	
牛顿L2	在计算过程中使用二阶导数。这可以改善模型的结果质量。
NewtonCosine (不能与 Lossguide 树生长策略一起使用)	
傻瓜L2	提供对L2估计的不同启发式
太阳能L2	
卫星L2	

知乎 @马东什么

catboost实现了xgboost的打分函数的形式——上图中的牛顿l2，也实现了原始的gbdt的打分函数的形式——上图的l2，除此之外，又创造了另外的四种启发式的打分函数——余弦、傻瓜、太阳能、卫星，哈哈哈哈哈这谷歌翻译哈哈哈哈哈，原图是

Score function	Description
L2	Use the first derivatives during the calculation.
Cosine (can not be used with the Lossguide tree growing policy)	
NewtonL2	Use the second derivatives during the calculation. T
NewtonCosine (can not be used with the Lossguide tree growing policy)	
LOOL2	Provide different heuristics of the L2 estimates
SolarL2	
SatL2	

知乎 @马东什么

目前暂时只找到了cos的定义：

$$Cosine = \frac{\sum w_i \cdot a_i \cdot g_i}{\sqrt{\sum w_i a_i^2} \cdot \sqrt{\sum w_i g_i^2}}$$

也是够奇怪。。。

因此简单说catboost继承了xgboost打分函数的基础上扩展了其它的一些打分函数来决定叶子是否进行分裂的。需要注意的一个地方：

$$L2 = - \sum_i w_i \cdot (a_i - g_i)^2$$

$$Cosine = \frac{\sum w_i \cdot a_i \cdot g_i}{\sqrt{\sum w_i a_i^2} \cdot \sqrt{\sum w_i g_i^2}}$$

catboostd的所有打分函数的定义都设计到这里的权重 w_i ，我一开始看的时候也是比较懵逼，这个权重是啥，不过后来看了catboost对行采样的定义之后才弄明白。这里就需要插入一个新的知识点了，这个细节很多很烦，后面具体详细描述，这里简单说一下：

啥是权重 w_i

https://catboost.ai/docs/concepts/algorithm-main-stages_bootstrap-options.htmlcatboost.ai

相关选项	描述	相关参数
贝叶斯	<p>模型的权重设置为以下值：</p> $w = a^f$, where: <ul style="list-style-type: none"> f 由 <code>bootstrap_type</code> 参数指定。 $a = -\log(\phi)$，其中 ϕ 从 <code>Uniform(0,1)</code> 独立生成。这等效于 a 从均匀分布中采样所有可能的值（请参阅 D. Rubin “贝叶斯统计推断”，1993 年，第 2 节）。 <p>① 注意：该贝叶斯权重仅用于正则化，而不是加权。</p>	<ul style="list-style-type: none"> 训练选项 训练单位
伯努利	<p>对应于随机森林增强（SFB，有关详细信息，请参阅本文）。对每个实例进行独立采样，以模拟 <code>subsample</code> 参数效果（对数据采样而非拆分）。所有样本实例均具有相同的权重。尽管最初建议将 SFB 用于正则化，但它几乎可以加快计算速度 $\left(\frac{1}{subsample}\right)$。</p>	<ul style="list-style-type: none"> 子采样 训练单位
MUS（仅 GPU 上支持）	<p>实现重要性采样算法，称为最小方差估计（MUS）。分裂的权重估计基于在叶子（由前次分裂提供）中抽取随机估计。其中抽取随机估计计算如下：</p> $g_i = \frac{\partial L(g_i, y)}{\partial g_i} \Big _{g_i = g_i}$	<ul style="list-style-type: none"> <code>mus_rng</code> 训练单位

我们常规意义上针对于随机森林或者gbdt的采样的概念是有放回抽样，catboost将这个概念进行了进一步的推广，它将我们常用的有放回抽样定义为样本权重服从伯努利分布，即样本的权重为0或者1，

$$Pr\{X = 1\} = p, Pr\{X = 0\} = 1 - p, 0 < p < 1.$$

p就是我们定义的采样比例了。这么做的好处在于我们可以通过替换伯努利分布来实现更多类型的bootstrap，比如经典的贝叶斯采样 **bayesian bootstrap**，这里后面再展开，因为展开要写好多，这里暂时简单理解为 随机权重法，就是不同样本的权重是随机生成的一堆数字，比如5个样本的权重分别为 [0.1,0.5,0.3,0.11,2]这种形式。

所以在一开始理解的时候把这个w当作1来看待就好了，在考虑样本采样的时候再去看他就行了。



优化方法：

- GBDT在优化时只使用了一阶导数信息，XGBoost在优化时使用了一、二阶导数信息。catboost可以由用户自由选择是使用一阶梯度还是一二阶梯度。

缺失值处理：

<https://catboost.ai/docs/concepts/algorithm-missing-values-processing.html>catboost.ai

首先是：

- 没有
- 浮点NaN值 
- 从文件加载或作为Python字符串时，以下字符串之一：
"
", "#N/A", "#N/AN/A", "#NA", "-1.#IND", "-1.#QNAN", "在", "-in", "1.#IND",
", "1.#QNAN", "N/A", "NA", "NULL", "NaN", "n/a", "nan", "null", "NAN", "Na", "
na", "Null", "none", "None", "-"
这是pandas中  默认缺失值列表的扩展版本。

知乎 @马东什么

其实和xgb与lgb的missing_value参数差不多就是指定啥是缺失值，感觉这个功能也比较鸡肋，一般对于这类值直接replace成nan就可以了。。。

支持以下模式处理缺失值:

- “Forbidden” —不支持缺少的值,它们的存在被解释为错误。
- “Min” (最小值) -缺失值作为特征的最小值 (小于所有其他值) 处理。确保在选择树时考虑将缺失值与所有其他值分开的拆分。
- “Max” (Max) -缺失值作为特征的最大值 (大于所有其他值) 处理。确保在选择树时考虑将缺失值与所有其他值分开的拆分。

默认处理模式为 Min。更改默认模式的方法取决于软件包:

知乎 @马东什么

catboost处理缺失值的方法很kaggle, 其实就和平常比赛的时候把缺失值替换为9999, -9999一个意思, 这里简单使用max和min其实并不是很好, 你在训练集指定的max和min可能在测试集存在更大的max值或者更小的min值, 这个功能做的没有xgboost方便, 至于效果好坏还真不好说, xgboost的稀疏处理也有自己的缺陷, 比如某个特征值缺失99%, 按照xgboost的稀疏感知, 仅仅在剩下1%的样本上决定分裂点, 然后缺失样本统一进入左或者右节点, 这样非常容易过拟合, 但是如果把缺失值赋予一个很大或者很小的值, 那么在当前特征缺失的样本都会被划分到单独的一个节点中从而继续接下来的分裂, 实际上类似于对于缺失和非缺失样本分开来进行建模, “缺失”的信息被模型所考虑。当然, 缺失值是做这种简单的插补还是放任不管对于模型的泛化性能没有什么严格的理论证明, 尝试过不同数据集, 效果也是好坏参半的。

防止过拟合:

- Catoost除了增加了正则项来防止过拟合,还支持行列采样的方式来防止过拟合。但是需要注意的是catboost对于bootstrap概念的推广。

https://catboost.ai/docs/concepts/algorithm-main-stages_bootstrap-options.htmlcatboost.ai

(ps: 对于catboost的列采样还是正常的采样, 参数为colsample_bylevel:

catboost.CatBoostRegressor				
num_classes	1	num_features	1000	1000
num_leaves	31	num_threads	4	4
num_trees	100	num_folds	5	5
num_valid	1	num_test	1	1

现在来看看行采样:

第一种模式, 贝叶斯采样:



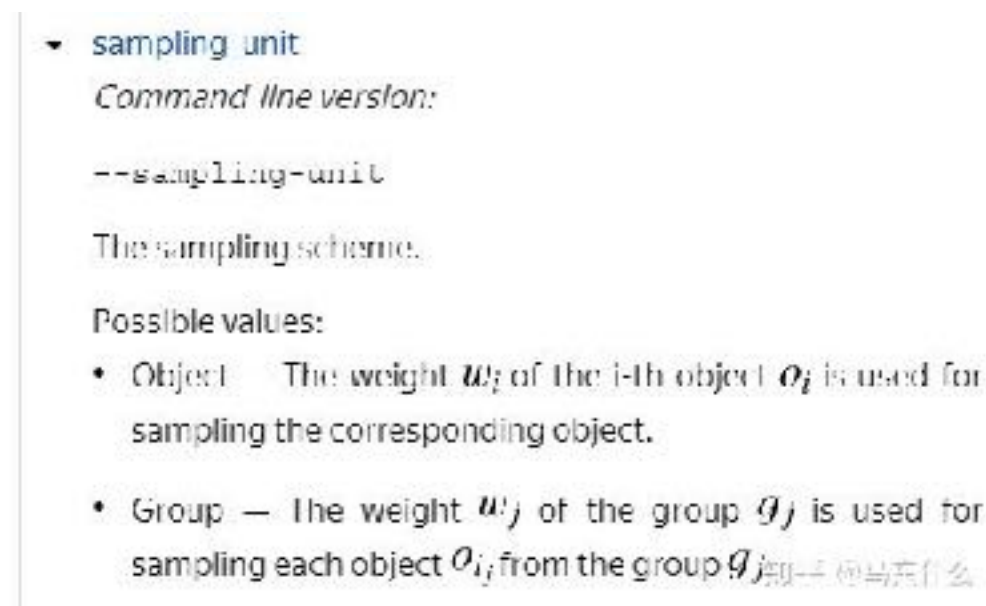
这里的t是bagging_temperatue，也是一个超参数，这里解释的很清楚了，

$a = -\log(\psi)$ ，其中 ψ 从Uniform [0,1]独立生成，

每一个样本的权重按照这样的方式随机生成，然后：

$$w = a^t,$$

难怪catboost速度不快。。。这么麻烦的吗。。。这里t越大则每一个样本的权重越小，相当于xgboost的subsample的程度越深。



另外sampling的模式还有两种。。。：

1、第一种 object模式就是常规的样本层面的sampling，这里catboost是针对每一个样本进行随机赋权；

2、第二种模式是group模式，针对与一阶梯度 g ，每一个 g 的group中对应的样本的权重相同，不同group的 g 的权重不同？？？？啥意思，没明白这里是怎么个操作法。。。。？？先留着

Catboost的基学习器

catboost与xgboost的level-wise以及lightgbm的leaf-wise不同，catboost的基学习器使用的是完全对称二叉树(实际上catboost实现了多种树的生长方式):

Possible values:

- **SymmetricTree** — A tree is built level by level until the specified depth is reached. On each iteration, all leaves from the last tree level are split with the same condition. The resulting tree structure is always symmetric.
- **Depthwise** — A tree is built level by level until the specified depth is reached. On each iteration, all non-terminal leaves from the last tree level are split. Each leaf is split by condition with the best loss improvement.
- **Lossguide** — A tree is built leaf by leaf until the specified maximum number of leaves is reached. On each iteration, non-terminal leaf with the best loss improvement is split.

知乎 @马东什么

树木生长政策。定义如何执行决策树的构建。

可能的值：

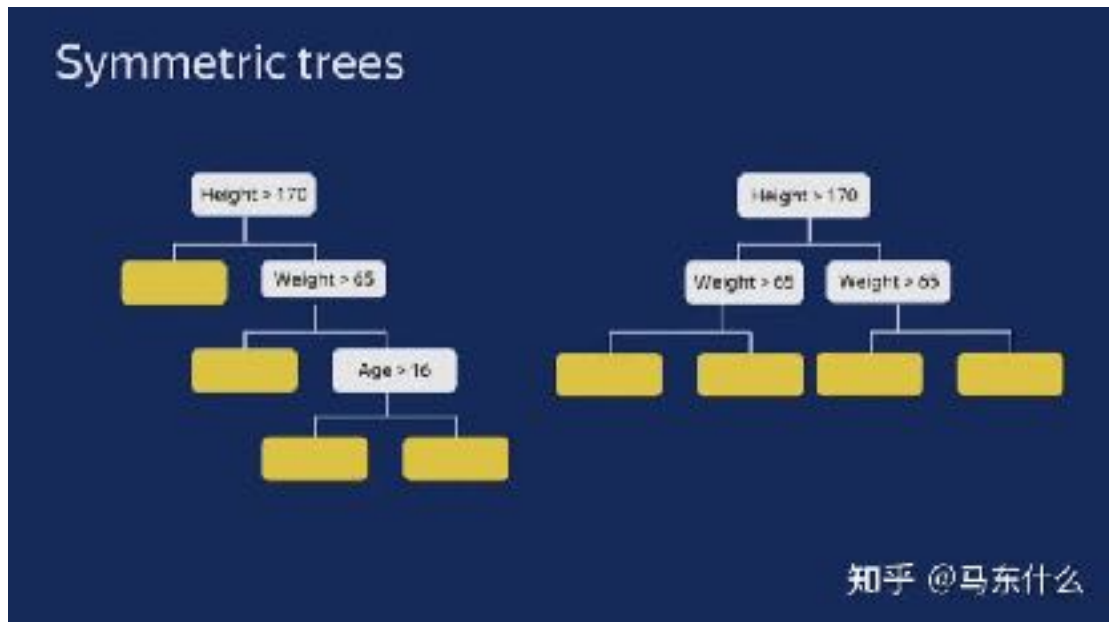
- SymmetricTree — 逐级构建树，直到达到指定的深度。在每次迭代中，以相同条件分割最后一棵树级别的所有叶子。生成的树结构始终是对称的。
- 深度 - 逐级构建一棵树，直到达到指定的深度。在每次迭代中，将拆分来自最后一棵树的所有非终端叶子。每片叶子均按条件分割，损失改善最佳。
- Lossguide- 逐叶构建一棵树，直到达到指定的最大叶数。在每次迭代中，将损失损失最佳的非终端叶子进行拆分。

❗ 注意。目前仅在训练和预测模式中支持“深度向导”和“损失指南”增长策略。不支持将它们用于模型分析（例如功能重要性和ShapValues）以及导出到不同的模型格式（例如 AppleCoreML，onnx和json）[如何忘记什么](#)

第一种就是上面提到的完全对称二叉树，概念可见（oblivious tree=symmetrictree）



oblivious tree在机器学习中有何用？www.zhihu.com



左图是lgb的leaf-wise的模式，右图是catboost的完全二叉树的形式，catboost称Catboost称对称树有利于避免overfit，增加可靠性，并且能大大加速预测等等。

这里的分裂过程实际上是这样的，我们以上图为例：

首先Height>170分裂，

1、则左节点A为height>170,右节点B为height<170；

2、此时我们有两个叶子节点A和B，正常按照xgb的做法，A继续在所有feature里research找分裂增益最大的叶子节点，B也是在所有feature里面找分裂增益最大的，比如A找到weight>65为分裂节点继续分裂，而B找到age>20为分裂节点继续分裂，这是传统的level-wise；而按照lgb的做法，A找到某个最优特征的最优分裂点，假设其增益gain为x1，而B找到的某个最优特征的最佳分裂点，假设其增益gain为x2，此时进行比较，如果x1>x2，则分裂A不分裂B，如果x1<x2则分裂B而不分裂A，此为leaf-wise；而catboost的做法和lgb有点类似，也是进行x1和x2的比较，但是比较完毕之后，是对A和B两个节点都使用B所得到的最优特征的最佳节点进行分裂。

然后是：

- Depthwise — A tree is built level by level until the specified depth is reached. On each iteration, all non-terminal leaves from the last tree level are split. Each leaf is split by condition with the best loss improvement.
- Lossguide — A tree is built leaf by leaf until the specified maximum number of leaves is reached. On each iteration, non-terminal leaf with the best loss improvement is split.

知乎 @马志什么

和xgb的参数设计是一毛一样的。。。

depthwise是xgb的默认生长方式也就是我们熟知的level-wise，lossguide则是lgb的生长方式，仅在增益最大的叶子生长，也就是我们熟知的leaf-wise。

catboost的模型

$|D| = \{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ ，集成模型的预测输出表示为：

$$\hat{y}_i = \phi(x_i) = \sum_{k=1}^K f_k(x_i) \quad (1)$$

其中， f_k 表示回归树， K 为回归树的数量。整个式(1)表示给定一个输入 x_i ，输出 K 颗回归树的预测值（即按照相应回归树的决策规则，所划分到的叶节点的权重）相加。

和xgb以及lgb的形式是一样的，总体形式就是gbdt表达式。

模型的学习

通常情况下，怎样去学习一个模型？

1. 定义目标函数，即损失函数以及正则项。
2. 优化目标函数。

$$L(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k) \quad (2)$$

$$\text{其中 } \Omega(f) = \gamma T - \frac{1}{2} \lambda \|w\|^2$$

知乎 @马东什么

这里后续的求解和xgb类似，唯一不同的是，用户可以通过超参数的方式指定是否进行二阶泰勒展开还是沿用原始gbdt仅使用一阶梯度。

树的生成

在决策树的生成中，我们用ID3、C4.5、Gini指数等指标去选择最优分裂特征、切分点(CART时)，Catoost同样定义了特征选择和切分点选择的指标：

Score function	Description
L2	Use the first derivatives during the calculation.
Cosine (can not be used with the Lossguide tree growing policy)	
NewtonL2	Use the second derivatives during the calculation. 1
NewtonCosine (can not be used with the Lossguide tree growing policy)	
LOOL2	Provide different heuristics of the L2 estimates
SolarL2	
SatL2	

知乎 @马东什么

前面已经提到过了，除了实现xgb的打分函数，catboost还实现了其它的打分函数的。

分裂查找算法

Possible values:

- **SymmetricTree** — A tree is built level by level until the specified depth is reached. On each iteration, all leaves from the last tree level are split with the same condition. The resulting tree structure is always symmetric.
- **Depthwise** — A tree is built level by level until the specified depth is reached. On each iteration, all non-terminal leaves from the last tree level are split. Each leaf is split by condition with the best loss improvement.
- **Lossguide** — A tree is built leaf by leaf until the specified maximum number of leaves is reached. On each iteration, non-terminal leaf with the best loss improvement is split.

翻译 @马东什么

前面已经详细说明过了，catboost实现了这三种分裂方式，默认使用的是完全对称二叉树的分裂方法。

其它的重要的改进：

写了半天发现没写重点。。。。：

1、类别特征 的处理（ordered ts）：

lightgbm底层对于高基数类别特征是用**梯度进行了编码**，关于lgb的类别特征的处理可见：

CSDN-专业IT技术社区-登录blog.csdn.net

而catboost的则是使用**统计特征**对类别进行编码，

[https://catboost.ai/docs/concepts/algorithm-main-stages_cat-to-numeric.html#algorithm-main-stages_cat-](https://catboost.ai/docs/concepts/algorithm-main-stages_cat-to-numeric.html#algorithm-main-stages_cat-to-numeric)

to-numericcatboost.ai	Type
	Borders
	Buckets
	BinarizedTargetMeanValue
	Counter

官方提供了这四种编码的方法，然后给的例子又是第二种编码方式。。。下面主要介绍默认的第二种编码方法：

1. CatBoost接受 组对象同性和模型值作为输入。

下表显示了此阶段的结果。

对象 #	f_1	f_2	...	f_n	功 能 值
1	2	40	...	七石	1
2	1	55	...	独立游戏	11
3	5	34	...	流行音乐	1
4	2	15	...	七石	0
5	4	53	...	七石	0
6	2	48	...	独立游戏	1
7	5	42	...	七石	1
...					

2. 载入文件中的行会随机随机抽取几次，生成多个随机排列。

下表显示了此阶段的结果。

对象 #	f_1	f_2	...	f_n	功 能 值
1	4	53	...	七石	0
2	3	55	...	独立游戏	0
3	2	40	...	七石	1
4	5	42	...	七石	1
5	5	14	...	流行音乐	1
6	2	48	...	独立游戏	1
7	2	15	...	七石	0
...					

3. 使用以下公式将所有分类特征值转换为数字：

$$avg_target = \frac{countInClass + prior}{totalCount + 1}$$

- $countInClass$ 是具有当前分类特征值的对象的标签值等于“1”的次数。
- $prior$ 是分子初始值。它由启动参数确定。
- $totalCount$ 是具有与当前对象匹配的分类特征值的对象总数 (直到当前对象) 什么。

注意，这里是随机shuffle，并且标签也跟着一起shuffle的；

在具有音乐流派的示例中， $j \in [1:3]$ 接受值“rock”，“pop”和“indie”，并且print设置为0.05。下表显示了此阶段的结果。

对象#	f_1	f_2	...	f_n	功 能 值
1	4	53	...	0.05	0
2	3	55	...	0.05	0
3	2	40	...	0.025	1
4	5	42	...	0.35	1
5	5	34	...	0.05	1
6	2	18	...	0.025	1
7	2	45	...	0.5125	0

知乎 @马东什么

以岩石为例，第一个岩石对应标签为0，则countinclass=0，totalcount=1-1=0；

第二个岩石对应标签为1，则countinclass=0（注意原始公式，countinclass是不包括当前样本的，都是用这个样本之前的样本进行编码的，类似于leave one out的思路），totalcount= 2-1=1，则根据公式计算可以得到0.025；

第三个岩石对应的标签为1，则countinclass=1，totalcount=3-1=2，则根据公式计算可以得到0.35

。。。。。。。

依此类推，就完成了使用buckets进行编码的方法，其它的方法没试过，不过编码方式api里写了，就不一个一个去尝试了。。。

和lgb类似可以通过参数：one_hot_max_size 来决定是否对类别特征进行编码操作，当类别数量超过one_hot_max_size之后才进行编码，否则默认是1vm的分割的；

gradient bias:

原文是首先介绍了prediction shift，然后介绍了为了解决prediction shift才诞生了order boosting：

4 Prediction shift and ordered boosting

4.1 Prediction shift

In this section, we reveal the problem of prediction shift in gradient boosting, which was neither recognized nor previously addressed. Like in case of TS, prediction shift is caused by a special kind of target leakage. Our solution is called *ordered boosting* and resembles the ordered TS method.

Let us go back to the gradient boosting procedure described in Section 2. In practice, the expectation in (2) is unknown and is usually approximated using the same dataset \mathcal{D} :

$$h^t = \arg \min_{h \in H} \frac{1}{n} \sum_{k=1}^n \left(-g^t(\mathbf{x}_k, y_k) - b(\mathbf{x}_k) \right)^2. \quad (6)$$

Now we describe and analyze the following chain of shifts:

1. the conditional distribution of the gradient $g^t(\mathbf{x}_k, y_k) \mid \mathbf{x}_k$ (accounting for randomness of $\mathcal{D} \setminus \{\mathbf{x}_k\}$) is shifted from that distribution on a test example $g^t(\mathbf{x}, y) \mid \mathbf{x}$;
2. in turn, base predictor h^t defined by Equation (6) is biased from the solution of Equation (2);
3. this, finally, affects the generalization ability of the trained model F^T .

As in the case of TS, these problems are caused by the target leakage. Indeed, gradients used at each step are estimated using the target values of the same data points the current model F^{t-1} was built on. However, the conditional distribution $F^{t-1}(\mathbf{x}_k) \mid \mathbf{x}_k$ for a training example $\mathbf{x}_k \in \mathcal{D}$ is shifted, in general, from the distribution $F^{t-1}(\mathbf{x}) \mid \mathbf{x}$ for a test example \mathbf{x} . We call this a *prediction shift*.

为了方便理解不考虑行列采样的问题，假设我们始终使用全量数据训练，则每一轮梯度的拟合（即每一棵树的训练）都是在全部的数据上进行的，作者认为，训练集上的条件分布：

$$g^t(\mathbf{x}_k, y_k) \mid \mathbf{x}_k$$

会和测试集上的条件分布：

$$g^t(\mathbf{x}, y) \mid \mathbf{x};$$

存在分布上的差异，从而导致gbdt在训练集上过拟合（分布差异是过拟合的本质原因）

Related work on prediction shift The shift of gradient conditional distribution $g^t(\mathbf{x}_t, y_t) \mid \mathbf{x}_t$ was previously mentioned in papers on boosting [3, 13] but was not formally defined. Moreover, even the existence of non-zero shift was not proved theoretically. Based on the out-of-bag estimation [2], Breiman proposed *iterated bagging* [3] which constructs a bagged weak learner at each iteration on the basis of “out-of-bag” residual estimates. However, as we formally show in Appendix E, such residual estimates are still shifted. Besides, the bagging scheme increases learning time by factor of the number of data buckets. Subsampling of the dataset at each iteration proposed by Friedman [13] addresses the problem much more heuristically and also only alleviates it.

在related word里面提到了，通过subsample可以在一定程度上缓解这种过拟合，本质是通过集成学习的方式，在不同的子空间训练出不同的基模型进行集成从而提高集成模型对未知分布数据的鲁棒性。

with $p = 1/2$ and $y = f^*(\mathbf{x}) = c_1 x^1 + c_2 x^2$. Assume we make $N = 2$ steps of gradient boosting with decision stumps (trees of depth 1) and step size $\eta = 1$. We obtain a model $F = F^2 = h^1 + h^2$. W.l.o.g., we assume that h^1 is based on x^1 and h^2 is based on x^2 , what is typical for $|c_1| > |c_2|$ (here we set some asymmetry between x^1 and x^2).

Theorem 1 1. If two independent samples \mathcal{D}_1 and \mathcal{D}_2 of size n are used to estimate h^1 and h^2 , respectively, using Equation (5), then $\mathbb{E}_{\mathcal{D}_1, \mathcal{D}_2} F^2(\mathbf{x}) = f^*(\mathbf{x}) + O(1/2^n)$ for any $\mathbf{x} \in \{0, 1\}^2$.
2. If the same dataset $\mathcal{D} = \mathcal{D}_1 = \mathcal{D}_2$ is used in Equation (5) for both h^1 and h^2 , then $\mathbb{E}_{\mathcal{D}} F^2(\mathbf{x}) = f^*(\mathbf{x}) - \frac{1}{n-1} c_2 (x^2 - \frac{1}{2}) + O(1/2^n)$.

This theorem means that the trained model is an unbiased estimate of the true dependence $y = f^*(\mathbf{x})$, when we use independent datasets at each gradient step [1]. On the other hand, if we use the same dataset at each step, we suffer from a bias $-\frac{1}{n-1} c_2 (x^2 - \frac{1}{2})$, which is inversely proportional to the data size n . Also, the value of the bias can depend on the relation f^* : in our example, it is proportional to c_2 . We track the chain of shifts for the second part of Theorem 1 in a sketch of the proof below, while the full proof of Theorem 1 is available in Appendix A.

Sketch of the proof. Denote by ξ_{st} , $s, t \in \{0, 1\}$, the number of examples $(\mathbf{x}_t, y_t) \in \mathcal{D}$ with $\mathbf{x}_t = (s, t)$. We have $h^1(s, t) = c_1 s + \frac{c_1 \xi_{st}}{\xi_{0t} + \xi_{1t}}$. Its expectation $\mathbb{E}(h^1(\mathbf{x}))$ on a test example \mathbf{x} equals $c_1 x^1 + \frac{c_1}{2}$. At the same time, the expectation $\mathbb{E}(h^1(\mathbf{x}_t))$ on a training example \mathbf{x}_t is different and equals $(c_1 x^1 + \frac{c_1}{2}) - c_2 (\frac{2x^2 - 1}{n-1}) + O(2^{-n})$. That is, we experience a prediction shift of h^1 . As a consequence, the expected value of $h^2(\mathbf{x})$ is $\mathbb{E}(h^2(\mathbf{x})) = c_2 (x^2 - \frac{1}{2}) (1 - \frac{1}{n-1}) + O(2^{-n})$ on a test example \mathbf{x} and $\mathbb{E}(h^1(\mathbf{x}) + h^2(\mathbf{x})) = f^*(\mathbf{x}) - \frac{1}{n-1} c_2 (x^2 - \frac{1}{2}) + O(1/2^n)$. □ 知乎 @马东什么

后面还定量计算出了bias的理论数值，这里就不详细深入了。

我个人的理解如下，我自己觉得更能说服自己哈哈哈：

原始的gbdt无行列采样，所有的base tree都在同一个完整的训练数据集上拟合，然而和rf不一样的是，gbdt中的tree都不是独立训练的，而是用上一轮的tree计算得到的负梯度为标签继续训练，这就导致了如果上一轮的tree计算产生了偏差，这个偏差会继续在下一轮累计，那么假设训练集和测试集的分布差异较大，则偏差会在base trees中疯狂累计，从而使得整个gbdt过度拟合训练集，通过行列采样可以在一定程度上缓解这个问题，catboost不仅仅在xgboost引入行列采样、引入树的正则化这种从超参数层面控制过拟合程度的方面开发，更是在tree训练的过程中采用ordered boosting的方式来缓解这种过拟合。

下面就开始介绍ordered boosting，首先我们需要知道：

catboost实际上是支持两种boosting模式的：

boosting_type	string	Boosting scheme. Possible values: <ul style="list-style-type: none">• Ordered – Usually provides better quality on small datasets, but it may be slower than the Plain scheme.• Plain – The classic gradient boosting scheme.	Depends on the processing unit type, the number of objects in the training dataset and the selected learning mode	CPU and GPU Only the Plain mode is supported for the MultiClass loss on GPU
---------------	--------	---	---	--

plain表示常规的gbdt的boosting模式，ordered模式，根据这里的描述，在小型的数据集上能够提供更好的结果，但是耗时。

CPU
Plain
GPU
<ul style="list-style-type: none">• Any number of objects, MultiClass or MultiClassOneVsAll mode: Plain• More than 50 thousand objects, any mode: Plain• Less than or equal to 50 thousand objects, any mode but MultiClass or MultiClassOneVsAll: Ordered

这里官网给的建议是样本不超过5万可以考虑ordered的boosting type。

需要特别提到的是，catboost每次生成新树的时候都会shuffle一下训练数据然后重新计算类别特征的编码值。。。

为什么验证数据集上的指标值有时会比训练数据集上的指标值更好？

发生这种情况是因为，针对训练和验证数据集，基于分类特征的自动生成的数字特征的计算方式有所不同：

- **训练数据集**：对于数据集中的每个对象，特征的计算方式都不同。对于每个第*i*个对象，基于来自第一个*i-1*个对象（在某些随机排列中的第一个*i-1*个对象）的数据来计算特征。
- **验证数据集**：对数据集中的每个对象均使用计算特征。对于每个对象，使用训练数据集中所有对象的数据计算特征。

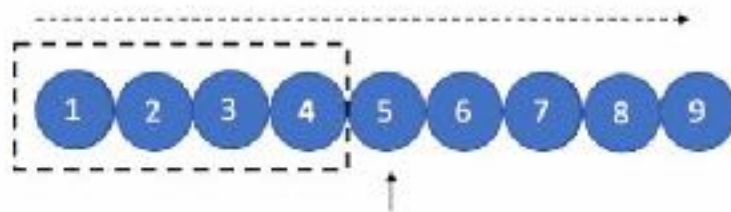
当根据来自训练数据集所有对象的数据计算特征时，它使用的信息比仅在数据集的一部分上计算的特征要多。因此，此功能更强大，更强大的功能可带来更好的预测值。

因此，由于验证数据集具有更强大的功能，因此验证数据集上的损失值可能会比训练数据集上的损失值更好。

另外需要注意的是，测试集的类别特征是用训练集的同一个类别特征的多个编码值进行平均所得；

还需要注意的是：

<https://towardsdatascience.com/getting-deeper-into-categorical-encodings-for-machine-learning-2312acd347c8>towardsdatascience.com

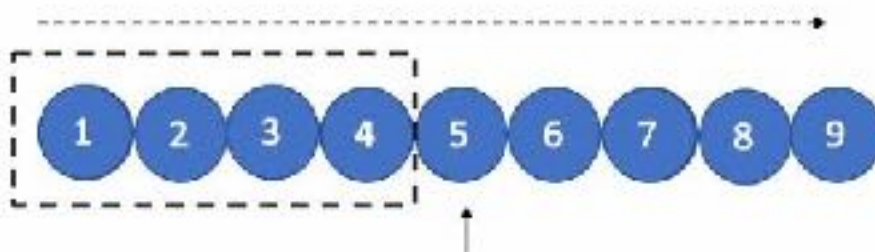


Running mean calculation.

Numbers are assigned randomly to each observation. Only 1-4 are used to find encoding for 5

But there is a problem in taking group-wise running averages. For the initial few rows in the randomized data, the mean encoding will have high variance as it only saw a few points from the history. But as it sees more data, the running average starts to get stable. This instability due to randomly chosen initial data points makes the initial part of dataset to have weak estimates. Catboost handles this problem by using more than one random permutations. It trains several models simultaneously, each one is trained on its own permutation. All the models share the same tree forest i.e the same tree structures. But leaf values of these models are different. Before building the next tree, CatBoost selects one of the models, which will be used to select tree structure. The structure is selected using this model, and is then used to calculate leaf values for all the models.

知乎 @马东什么



Running mean calculation.

Numbers are assigned randomly to each observation. Only 1-4 are used to find encoding for 5

但是，采用分组方式的运行平均值存在一个问题。对于随机数据中的最初几行，平均编码将其具有较高的方差，因为它仅从历史记录中看到了一些点。但随着看到更多数据，运行平均值开始趋于稳定。由于随机选择的初始数据点导致的这种不稳定性使数据集的初始部分具有较弱的估计。Catboost通过使用多个随机排列来解决此问题。它同时训练多个模型，每个模型都被其自身的排列进行训练。所有模型共享相同的树杯，即相同的树结构。但是这些模型的叶子值不同。在构建下一棵树之前，CatBoost将选择其中一个模型，这些模型将用于选择树结构。使用此模型选择结构。

在很多地方看到，默认的shuffle次数是四次，并且随机选择其中一个shuffle的编码结果进行分裂。cabooost的官网找遍了也没找到相关的介绍，wtf！！！！你倒是把说明文档好好整整啊！！

关于ordered boosting type，是这样的：

Algorithm 1: Updating the models and calculating model values for gradient estimation

input : $\{(\mathbf{X}_k, Y_k)\}_{k=1}^n$ ordered according to σ , the number of trees I ;

```

1  $M_i \leftarrow 0$  for  $i = 1..n$ ;
2 for iter  $\leftarrow 1$  to  $I$  do
3   for  $i \leftarrow 1$  to  $n$  do
4     for  $j \leftarrow 1$  to  $i-1$  do
5        $g_j \leftarrow \frac{d}{ds} \text{Loss}(y_i, a) \big|_{a=M_i(\mathbf{X}_j)}$ ;
6        $M \leftarrow \text{LearnOneTree}(\mathbf{X}_j, g_j)$  for  $j = 1..i-1$ ;
7        $M_i \leftarrow M_i + M$ ;
8 return  $M_1, \dots, M_n; M_1(\mathbf{X}_1), M_2(\mathbf{X}_2), \dots, M_n(\mathbf{X}_n)$ 

```

知乎 @马东什么

在CatBoost中，我们生成训练数据集的s个随机排列。采用多个随机排列是为了增强算法的鲁棒性，这在前面的Ordered TS当中对于类别型特征的处理有介绍到：针对每一个随机排列，计算得到其梯度，为了与Ordered TS保持一致，这里的排列与用于计算Ordered TS时的排列相同。(也就是说，ordered ts和ordered boosting是使用的同样的shuffle)

我们观察上面的伪代码：

—
 σ

表示一次shuffle，默认是使用4次shuffle，因此上述伪代码的流程会走4遍；

—
 $\{(\mathbf{X}_k, Y_k)\}_{k=1}^n$

表示所有的训练样本，注意这里的n表示的是样本的数量

—
the number of trees I

I 表示tree的数量，根据用户定义的 $n_estimators$ 来的。

```
 $M_i \leftarrow 0 \text{ for } i = 1..n;$ 
```

初始对所有的样本 n ，每一个样本都赋一个对应的 $M_i=0$ 。

```
for  $iter \leftarrow 1$  to  $I$  do
```

表示，针对每一个样本的建树过程都遵循接下来的流程。

```
for  $i \leftarrow 1$  to  $n$  do
```

对每一个样本都进行接下来的操作：

```
for  $j \leftarrow 1$  to  $i - 1$  do
```

```
     $g_j \leftarrow \frac{d}{da} Loss(y_j, a)|_{a=M_i(\mathbf{x}_j)};$ 
```

```
     $M \leftarrow LearnOneTree((\mathbf{X}_j, g_j) \text{ for } j = 1..i - 1);$ 
```

```
     $M_i \leftarrow M_i + M;$ 
```

这里举个例子详细解释一下，catboost的ordered boost type在一些地方被称为引入了online learning的思路，啥意思呢，假设我们要计算 x_5 的负梯度，则我们使用 x_1, x_2, x_3, x_4 训练一棵树，用这棵树predict出 x_5 的预测值然后和真实值带入负梯度计算公式算出负梯度，从而作为 x_5 的负梯度 g 作为下一轮的标签。如果要计算 x_6 的负梯度，则使用 x_1, x_2, x_3, x_4, x_5 按照上面的方法如法炮制。

显然，这种计算方式将导致非常大的计算量，因此catboost内部做了一些改进，默认情况下，catboost只训练 $\log(\text{num_of_datapoints})$ 个模型，而不是为每个数据点训练不同的模型。

意思是：

*在第一个数据点上训练的模型用于计算第二个数据点的残差。

*在前两个数据点上经过训练的另一个模型用于计算第三和第四数据点的残差

*在前四个数据点上经过训练的另一个模型用于计算第5, 6, 7, 8个数据点的残差

依次类推

这里的online learning体现在了样本的顺序上，实际上对于普通的无序的表格数据，其顺序是随机定义的，我们前面说过计算类别特征的ordered ts的时候要随机shuffle，这里的ordered boosting也是随机shuffle之后按照shuffle之后的结果作为其顺序排列，比如小红 小明 小王三个样本shuffle之后变成小王 小明 小红，则样本的顺序就是小王是x1，小明是x2，小红是x3。然后再shuffle一次变成 小明，小红，小王，则样本的顺序是小明x1，小红x2，小王x3。

实际上，CatBoost将给定的数据集划分为随机排列，并对这些随机排列应用有序增强。默认情况下，CatBoost创建四个随机排列。有了这种随机性，我们可以进一步停止过度拟合我们的模型。我们可以通过调整参数bagging_temperature来进一步控制这种随机性。这是您在其他增强算法中已经看到的

应用场景

作为GBDT框架内的算法，GBDT、XGBoost、LightGBM能够应用的场景CatBoost也都适用，并且在处理类别型特征具备独有的优势。

优缺点

优点

- 能够处理类别特征
- 能够有效防止过拟合
- 模型训练精度高
- 调参时间相对较多

缺点

- 对于类别特征的处理需要大量的内存和时间
- 不同随机数的设定对于模型预测结果有一定的影响

Tree系列

1、介绍随机森林和GBDT的区别，为什么Bagging降方差，Boosting降偏差

GBDT和随机森林的相同点：

- 1、都是基于tree的集成模型框架；
- 2、最终的推理都是所有的tree来共同完成的；

GBDT和随机森林的不同点：

- 1、组成随机森林的树可以是分类树，也可以是回归树；而GBDT只由回归树组成（因为梯度是连续值无法使用分类树）

- 2、随机森林使用了行列采样来划分不同的数据集进行tree的训练，每一颗tree的训练都是完全独立的，因此便于并行；而gbdt每一颗tree的训练都是上一轮的tree拟合得到的负梯度，因此无法在tree的层面并行；
- 3、对于最终的输出结果而言，随机森林采用多数投票、或者直接进行概率平均；而GBDT则是将所有结果加权累加起来
- 4、随机森林对异常值不敏感，GBDT对异常值很敏感
- 5、随机森林的tree的深度一般比gbdt深，因为bagging减少方差难以降低偏差因此对于基学习器的拟合能力是有较高的要求的；
- 6、随机森林是通过减少模型方差提高性能，GBDT是通过减少模型偏差提高性能，而后续的xgb lgb等对gbdt系列算法引入了行列采样，不但可以降低方差也可以降低偏差

2、为什么Bagging降方差，Boosting降偏差

Bagging对样本重采样，对每一重采样得到的子样本集训练一个模型，最后取平均。由于子样本集的相似性以及使用的是同种模型，因此各模型有近似相等的偏差s和方差。

由于 $E[\frac{\sum X_i}{n}] = E[X_i]$ （这里 X_i 表示子模型的偏差），所以bagging后的偏差和单个子模型的接近，一般来说不能显著降低bias。另一方面，若各子模型

独立，则有 $Var(\frac{\sum X_i}{n}) = \frac{Var(X_i)}{n}$ ，此时可以显著降低方差。若各子模型

完全相同，则 $Var(\frac{\sum X_i}{n}) = Var(X_i)$ ，此时完全不降低方差；

。bagging方法得到的各子模型是有一定相关性的，属于上面两个极端状况的中间态，因此可以一定程度降低variance。为了进一步降低variance，Random forest通过随机选取特征来进一步降低子模型之间的相似性从而进一步降低方差；

boosting从整体上划分为adaboost家族和gbm家族，adaboost家族不仅可以降低偏差也可以降低方差，之所以能降低方差是因为adaboost系列包括离散、real、gentle adaboost和logitboost都是通过不同的策略来对样本进行reweight，reweight实际上是一种广义的bagging手段，bagging本身可以看作是一种0-1加权的方法，因此实际上adaboost的这种权重自动调整方法也可以起到降低方差的效果，具体可见：

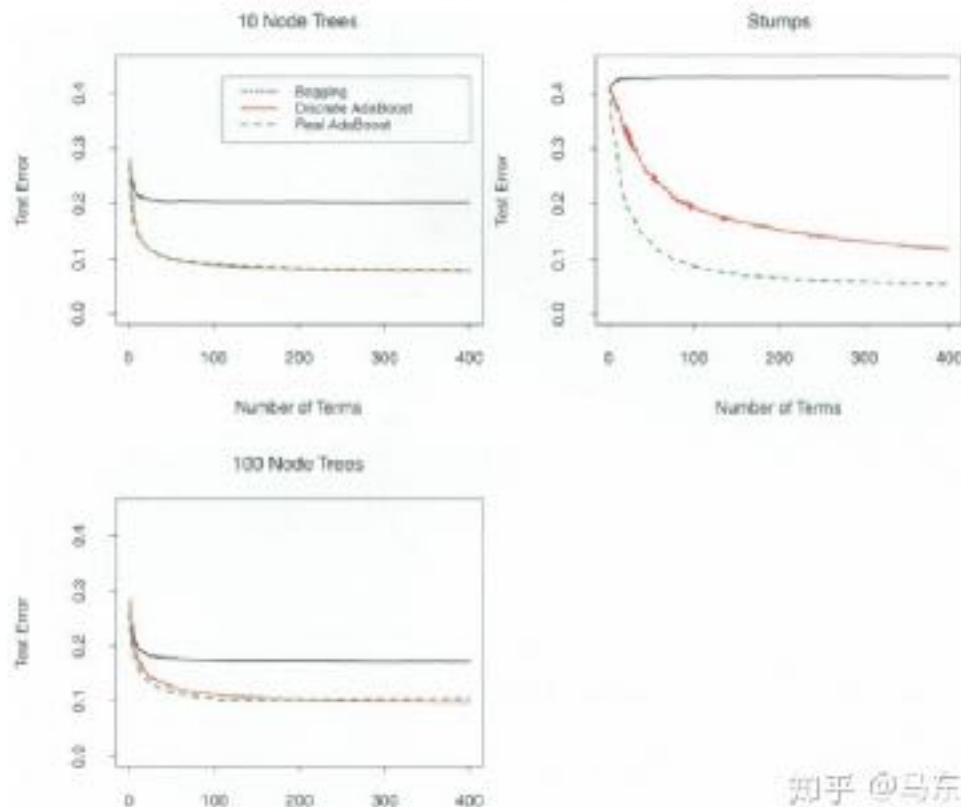
SPECIAL INVITED PAPER

ADDITIVE LOGISTIC REGRESSION: A STATISTICAL VIEW OF BOOSTING

BY JEROME FRIEDMAN,¹ TREVOR HASTIE^{2,3} AND
ROBERT TIBSHIRANI^{2,4}

Stanford University

Boosting is one of the most important recent developments in classification methodology. Boosting works by sequentially applying a classification algorithm to reweighted versions of the training data and then taking a weighted majority vote of the sequence of classifiers thus produced. For many classification algorithms, this simple strategy results in dramatic improvements in performance. We show that this seemingly mysterious phenomenon can be understood in terms of well-known statistical principles, namely additive modeling and maximum likelihood. For the two-class problem, boosting can be viewed as an approximation to additive modeling on the logistic scale using maximum Bernoulli likelihood as a criterion. We develop more direct approximations and show that they exhibit nearly identical results to boosting. Direct multiclass generalizations based on multinomial likelihood are derived that exhibit performance comparable to other recently proposed multiclass generalizations of boosting in most situations, and far superior in some. We suggest a minor modification to boosting that can reduce computation, often by factors of 10 to 50. Finally, we apply these insights to produce an alternative formulation of boosting decision trees. This approach, based on best-first truncated tree induction, often leads to better performance, and can provide interpretable descriptions of the aggregate decision rule. It is also much faster computationally, making it more suitable to large-scale data mining applications. 知乎 @马东什么



知乎 @马东什么

大神friedman做过实验测试adaboost和bagging的效果，只能说rf中除了引入样本采样也引入了特征采样可能对于方差的降低效果更好；

而对于gbdt来说，常用的xgboost和lightgbm也引入了行列采样的功能了，在boosting的框架上引入了rf的思想，其泛化性能在实践中也取得了很好的成绩。感觉，boosting减少偏差不减少方差的说法略微有失偏颇

4.介绍XGB对GBDT的提升，LGB对XGB的提升，以及既然使用了LGB为什么还要使用XGB

前面已经写过了；

5.为什么要大量使用树模型，有什么优势

1.高性能的开源实现，相对于nn来说简单的超参数设置，开箱即用型很强；

2.泛化性能强，低方差 低偏差；

3.XGB如何处理缺失值，LGB的差加速和直方图算法的底层代码是 否有过了解

Xgb处理缺失值，在未缺失的样本上正常分裂，然后将缺失样本分别放入左枝和右枝进行分裂增益的计算，那边增益大就直接将所有缺失样本放到那边然后继续分裂；

5、GBDT的原理？和Xgboost的区别联系？

上面介绍过了

6.adaboost和gbdt的区别联系？

和AdaBoost一样，Gradient Boosting也是重复选择一个表现一般的模型并且每次基于先前模型的表现进行调整。不同的是，AdaBoost是根据base estimator的错误来对样本进行reweight来定位模型的不足而Gradient Boosting是通过算梯度（gradient）来定位模型的不足。因此相比AdaBoost，Gradient Boosting可以使用更多种类的目标函数。

树模型如何调参

Max depth，tree的数量，行列采样的比例，是比较立竿见影的超参数；

树模型如何剪枝？

前面提到过了，预减枝和后减枝，目前常用的还是通过超参数的设置来进行预减枝叶较多，后减枝叶的计算代价太大，对于常用的tree的集成学习而言是比较耗费时间的；

决策树有哪些划分指标？区别与联系？

信息增益、信息增益率、gini指数以及xgboost时代的gain的计算公式

决策树怎么分裂的

Id3 根据信息增益的大小进行分裂，这种方式会使得tree偏向于选择类别数量较多的类别特征，c.45采用启发式的方法，先选择信息增益在平均水平之上的然后选择信息增益率大的特征进行分裂，cart则是根据gini指数，选择分裂之后纯度最大的特征进行分裂；

手推xgboost?

可见上面的原理介绍部分

xgboost, rf, lr优缺点场景

这种问题感觉就没啥好讲的。。。就是每个算法的题目糅杂在一起而已；

5.GBDT是否只能用CART树，GBDT中残差

计算公式

应该所gbdt只能使用回归树，xgboost的base tree严格来说已经不属于cart tree了，因为不是使用gini进行分裂的；

残差计算公式（残差其实是mse的一阶梯度） $\frac{1}{2}*(y-f(x))^2$ 求导之后得到 $y-f(x)$

4.GBDT的原理，如何做分类和回归

上面已经介绍过了

5.随机森林的随机体现在哪方面

每一个基tree所训练的数据子集都是不一样的，具体是通过对样本和特征进行采样来实现的；

7.GBDT+LR是怎么做的？

假设gbdt训练完毕有1000个叶子节点，对所有样本进行预测，样本落在其中某个叶子节点则可以知道该样本符合某种符合规则，例如性别男，年龄大于20岁并且年收入小于20万，对所有样本进行预测得到其叶节点的index，然后进行onehot展开再进行lr，使得lr能够捕捉到潜在的非线性关系；

23.如果有一万个地理坐标，转换成1-10000的数，可以用决策树么？

当然不行，除非地理坐标和1-10000之间的大小关系是一一对应的，如果是一一对应的的话那么其实差别不大，tree的分裂时基于数据的相对排序性而与其绝对大小无关，例如 0 1 2 3和0 100 200 300在同一个问题的前提下，对于tree来说其分裂时没有差别的；

25、CART分类树和ID3以及C4.5有什么区别？

Cart tree使用gini指数作为分裂标准并且是标准的二叉树；

AdaBoost是如何改变样本权重

离散型：

Discrete AdaBoost [Freund and Schapire (1996b)]

1. Start with weights $w_i = 1/N, i = 1, \dots, N$.
2. Repeat for $m = 1, 2, \dots, M$:
 - (a) Fit the classifier $f_m(x) \in \{-1, 1\}$ using weights w_i on the training data.
 - (b) Compute $\text{err}_m = E_w[1_{(y \neq f_m(x))}]$, $c_m = \log((1 - \text{err}_m)/\text{err}_m)$.
 - (c) Set $w_i \leftarrow w_i \exp[c_m 1_{(y_i \neq f_m(x_i))}]$, $i = 1, 2, \dots, N$, and renormalize so that $\sum_i w_i = 1$.
3. Output the classifier $\text{sign}[\sum_{m=1}^M c_m f_m(x)]$.

知乎 @马东什么

通过样本的错分概率来对权重进行reweight，预测错误的样本权重上升，否则下降；

Real AdaBoost

1. Start with weights $w_i = 1/N$, $i = 1, 2, \dots, N$.
 2. Repeat for $m = 1, 2, \dots, M$:
 - (a) Fit the classifier to obtain a class probability estimate $p_m(x) = \hat{P}_w(y = 1|x) \in [0, 1]$, using weights w_i on the training data.
 - (b) Set $f_m(x) \leftarrow \frac{1}{2} \log p_m(x)/(1 - p_m(x)) \in R$.
 - (c) Set $w_i \leftarrow w_i \exp[-y_i f_m(x_i)]$, $i = 1, 2, \dots, N$, and renormalize so that $\sum_i w_i = 1$.
 3. Output the classifier $\text{sign}[\sum_{m=1}^M f_m(x)]$.
-

Real adaboost则是通过预测概率值的比值来进行权重的reweight的，预测错误的样本权重上升，否则下降

GBDT分类树拟合的是什么？

拟合的时分类损失函数的负梯度；

3.问了随机森林有了解吗？知道里面的有

放回的采样方法吗？

了解，知道，rf用的有放回。

GBDT构造单棵决策树的过程？

Xgboost之前的gbdt其单个cart tree的训练是完全独立的一个训练过程，根据分裂之后gini的大小进行分裂直到达到预先设定的结束条件或者叶节点完全是一类样本从而结束分裂；

而xgboost的base tree其每一步分裂都是根据：

$$Gain = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma$$

这里的分裂增益公式来进行局部最优的分裂。=，直到达到余弦设定的结束条件或者叶节点完全是一类样本从而结束分裂；

3.GBDT和Xgboost的区别

前面提到过了

XGBoost的其参数的意义和调参的经验

xgboost有哪些参数？

- 学习率 eta：学习率越小，迭代次数越多。
- 最小孩子权重 min-child-weight：控制叶子结点中二阶导数和的最小值，即样本的数量越少（由于h大约均在0.01附近），越容易过拟合
- 最大深度 max_depth
- 最大叶子结点数 max_leaf_node
- 后剪枝参数gamma
- L2参数lambda
- L1参数alpha (控制模型复杂度)

-样本随机采样 subsample；列采样比例 colsample_bytree

2. Gbdt怎么并行，树个数和深度怎么选择

Gbdt不是在tree的层面上并行而是在特征的层面分裂，例如有100个特征，我们可以通过并行，例如每一个子进程处理10个特征的分裂，则可以分到10个子进程成进行并行计算

4.树模型的特征选择中除了信息增益、信息增益比、基尼指数这三个外，还有哪些？Xgboost的gain的计算公式

$$Gain = \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma$$

rf说一下，rf树特别多会发生什么，rf和

gbdt哪个深一些，为什么

rf深，gdbt关注偏差减小，对于base tree的拟合能力要求不高，rf关注方差减小，对于base tree的拟合能力要求较高，而max_depth是决定其性能的关键因素；

id3复杂度是多少，怎么优化呢（不会优化，，，）

3.XGB特征重要性程度是怎么判断的？

Xgboost的衡量特征重要性的手段有很多：

(**gain 增益**意味着相应的特征对通过对模型中的每个树采取每个特征的贡献而计算出的模型的相对贡献。与其他特征相比，此度量值的较高值意味着它对于生成预测更为重要。

cover 覆盖度量指的是与此功能相关的观测的相对数量。例如，如果您有100个观察值，4个特征和3棵树，并且假设特征1分别用于决策树1，树2和树3中10个，5个和2个观察值的叶结点;那么该度量将计算此功能的覆盖范围为 $10+5+2 = 17$ 个观测值。这将针对所有决策树结点进行计算，并将17个结点占总的百分比表示所有功能的覆盖指标。

freq 频率 (频率)是表示特定特征在模型树中发生的相对次数的百分比。在上面的例子中，如果feature1发生在2个分裂中，1个分裂和3个分裂在每个树1，树2和树3中;那么特征1的权重将是 $2 \times 1 \times 3 = 6$ 。特征1的频率被计算为其在所有特征的权重上的百分比权重。)

4.XGB很容易理解它的回归和二分类，如何理解多分类呢？

Xgb的多分类实际上是有两种模式的：

为什么xgb, lgb对稀疏矩阵处理不好

tree系列对于太过稀疏的特征矩阵都无法进行良好的处理，主要是因为：

1. 可能无法在这个类别特征上进行切分。使用one-hot coding的话，意味着在每一个决策节点上只能用 one-vs-rest (例如是不是狗，是不是猫，等等) 的切分方式。当特征维度高时，每个类别上的数据都会比较少，这时候产生的切分不平衡，切分增益 (split gain) 也会很小 (比较直观的理解是，不平衡的切分和不切分几乎没有区别)。

2、在不加限制的情况下，tree会一直在高维的稀疏特征中生长，从而像左图这样一直分裂下去，当叶子节点的样本数量很小的时候，我们难以直接就根据叶子节点的输出来判定样本的类别或者是样本的回归标签值，举一个简单的例子，加入某个叶节点一共就5个样本，4个正样本，1个负样本，那么我们可以直接判定落在这个叶子节点上的样本是正样本的概率为0.8吗？答案是不能，统计值本身就是建立在大量样本的情况下才有效，少量样本的统计特征相对于全量数据的统计特征存在严重的偏差，在样本数量很小的情况下，概率值是没有意义的。

xgb和lgb的树是怎么分裂的

上面的原理已经说过

XGBoost工程方面的改进有哪些？

其他优化方法

- **稀疏值处理 (Sparsity-aware Split Finding)** 。实际工程中一般会出现输入值稀疏的情况。比如数据的缺失、one-hot编码都会造成输入数据稀疏。论文中作者提出了关于稀疏值的处理，思路是：对于缺失数据让模型自动学习默认的划分方向。算法具体的方法如下：

Algorithm 3: Sparsity-aware Split Finding

Input: I , instance set of current node

Input: $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$

Input: d , feature dimension

Also applies to the approximate setting, only collect statistics of non-missing entries into buckets

$\text{gain} \leftarrow 0$

$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$

for $k = 1$ **to** m **do**

// enumerate missing value goto right

$G_L \leftarrow 0, H_L \leftarrow 0$

for j **in** $\text{sorted}(I_k, \text{ascend order by } x_{jk})$ **do**

$G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$

$G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$

$\text{score} \leftarrow \max(\text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

end

// enumerate missing value goto left

$G_R \leftarrow 0, H_R \leftarrow 0$

for j **in** $\text{sorted}(I_k, \text{descent order by } x_{jk})$ **do**

$G_R \leftarrow G_R + g_j, H_R \leftarrow H_R + h_j$

$G_L \leftarrow G - G_R, H_L \leftarrow H - H_R$

$\text{score} \leftarrow \max(\text{score}, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

end

end

Output: Split and default directions with max gain

从算法中可以看出，作者采用的是在每次的切分中，让缺失值分别被切分到左节点以及右节点，通过计算得分值比较两种切分方法哪一个更优，则会对每个特征的缺失值都会学习到一个最优的默认切分方向。乍一看这个算法会多出相当于一倍的计算量，但其实不是的。因为在算法的迭代中只考虑了非缺失值数据的遍历，缺失值数据直接被分配到左右节点，所需要遍历的样本量大大减小。

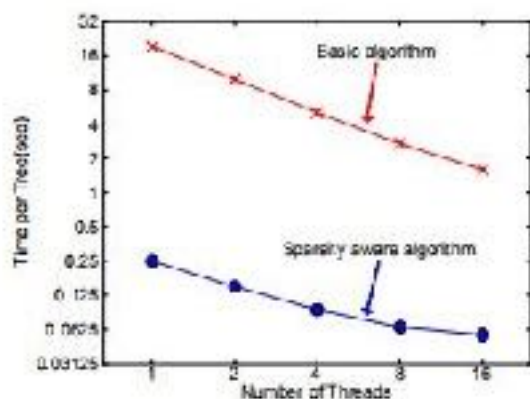


Figure 5: Impact of the sparsity aware algorithm on Allstate-10K. The dataset is sparse mainly due to one-hot encoding. The sparsity aware algorithm is more than 50 times faster than the naive version that does not take sparsity into consideration.

作者通过在Allstate-10K数据集上进行了实验，从结果可以看到稀疏算法比普通算法在处理数据上快了超过50倍。

- **分块并行 (Column Block for Parallel Learning)**。在树生成过程中，需要花费大量的时间在特征选择与切分点选择上，并且这部分时间中大部分又花费在了对特征值得排序上。那么怎么样减小这个排序时间开销呢？作者提出通过按特征进行分块并排序，在块里面保存排序后的特征值及对应样本的引用，以便于获取样本的一阶、二阶导数值。具体方式如图：

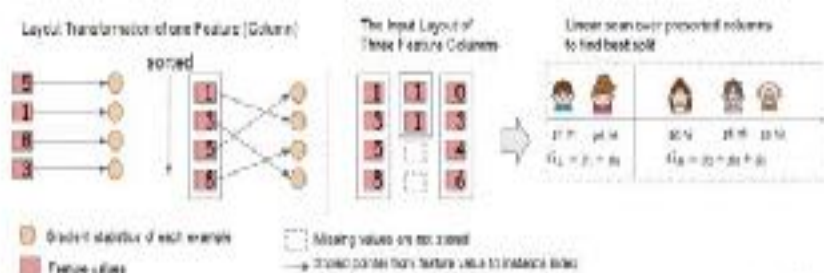


Figure 8: Block structure for parallel learning. Each column in a block is sorted by the next preceding feature value. A linear scan over one column in the block is sufficient to enumerate all the split points.

通过顺序访问排序后的块遍历样本特征的特征值，方便进行切分点的查找。此外分块存储后多个特征之间互不干涉，可以使用多线程同时对不同的特征进行切分点查找，即特征的并行化处理。注意到，在顺序访问特征值时，访问的是一块连续的内存空间，但通过特征值持有的索引（样本索引）访问样本获取一阶、二阶导数时，这个访问操作

访问的内存空间并不连续，这样可能造成cpu缓存命中率低，影响算法效率。那么怎么解决这个问题呢？缓存访问 **Cache-aware Access**。

- **缓存访问 (Cache-aware Access)**。为了减小非连续内存的访问带来缓存命中率低问题，作者提出了缓存访问优化机制。解决思路是：既然是非连续内存访问带来问题，那么去掉非连续内存访问就可以解决。那么怎么能去掉非连续内存空间的访问呢？**转非连续为连续---缓存预取**。即提起将要访问的非连续内存空间中的梯度统计信息（一阶、二阶导数），放置到连续的内存空间中。具体的操作上就是为每个线程在内存空间中分配一个连续的buffer缓存区，将需要的梯度统计信息存放在缓冲区中。这种方式对数据量大的时候很有用，因为大数据量时，不能把所有样本都加入到内存中，因此可以动态的将相关信息加入到内存中。

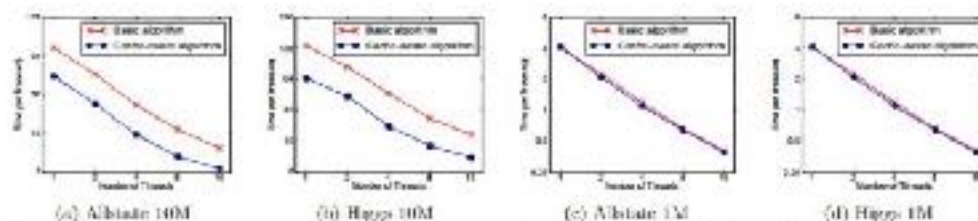
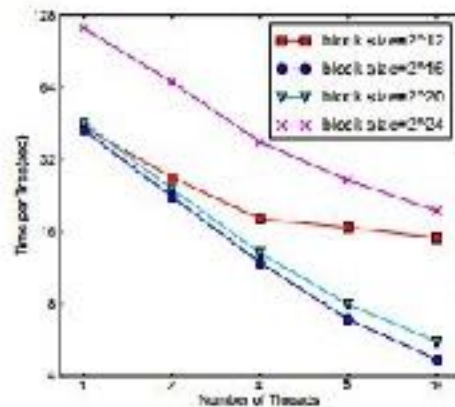
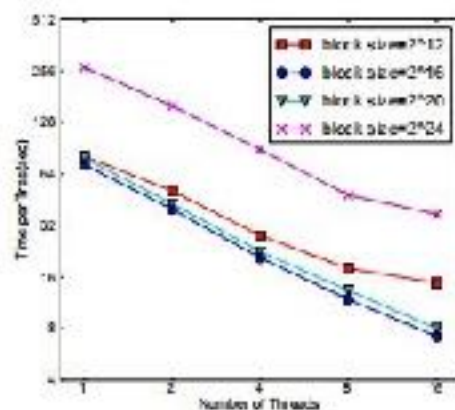


Figure 7: Impact of cache-aware prefetching in exact greedy algorithm. We find that the cache-aware effect impacts the performance on the large datasets (10 million instances). Using cache aware prefetching improves the performance by factor of two when the dataset is large.

上图给出了在Higgs数据集上使用缓存访问和不使用缓存访问的式样对比。可以发现在数据量大的时候，基于精确的贪心算法使用缓存预取得处理速度几乎是普通情况下的两倍。那么对于block块应该选择多大才合理呢？作者通过实验证明选择每个块存放 一个样本时效率最高。



(a) Allstate 10M



(b) Higgs 10M

Figure 9: The impact of block size in the approximate algorithm. We find that overly small blocks results in inefficient parallelization, while overly large blocks also slows down training due to cache misses.

- **"核外"块计算 (Blocks for Out-of-core Computation)**。当数据量非常大的时候我们不能把所有数据都加载内存中，因为装不下。那么就必须的将一部分需要加载进内存的数据先存放在硬盘中，当需要时在加载进内存。这样操作具有很明显的瓶颈，即硬盘的IO操作速度远远低于内存的处理速度，那么肯定会存在大量等待硬盘IO操作的情况。针对这个问题作者提出了“核外”计算的优化方法。具体操作为，将数据集分成多个块存放在硬盘中，使用一个独立的线程专门从硬盘读取数据，加载到内存中，这样算法在内存中处理数据就可以和从硬盘读取数据同时进行。为了加载这个操作过程，作者提出了两种方法。
- **块压缩 (Block Compression)**。论文使用的是按列进行压缩，读取的时候用另外的线程解压。对于行索引，只保存第一个索引值，然后用16位的整数保存与该block第一个索引的差值。作者通过测试

在block设置为 个样本大小时，压缩比率几乎达到26% ~ 29%（貌似没说使用的是什么压缩方法.....）。

- **块分区（Block Sharding）**。块分区是将特征block分区存放在不同的硬盘上，以此来增加硬盘IO的吞吐量。
- **防止过拟合**。从XGBoost的模型上可以看到，为了防止过拟合加入

了两项惩罚项 γT 、 $\frac{1}{2}\lambda||w||^2$ ，除此之外XGBoost还有另外两个防止过拟合的方法。**1.学习率**。和GBDT一样XGBoost也采用了学

习率（步长）来防止过拟合，表现为： $\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + \eta f_t(x_i)$ ，其中 就是学习率，通常取0.1。**2.行、列采样**。和随机森林一样XGBoost支持对样本以及特征进行采样，取采样后的样本和特征作为训练数据，进一步防止过拟合。

XGBoost原理，与GBDT的区别，XGBoost用二阶导数有什么优势？

要点：前两个问题很基本了，简单说Boost原理，区别提到二阶导数和正则项就够了。

至于二阶导数的优势主要在能自定义损失函数。

XGBoost和LightGBM、CatBoost主要区别？

要点1：XGBoost中，树是按层生长的（平衡的），同一层所有节点都做分裂，叫做Level-wise tree growth，最后剪枝。优点是过一次数据可以同时分裂同一层的叶子，比较好做多线程优化，但是比较低效，比如有些同层叶子增益很低实际没必要分裂。

要点2：LightGBM是不平衡的，叫做 Leaf-wise，就是每次从当前所有的叶子中，找到增益最大的一个叶子，然后分裂。缺点是可能长出比较深的决策树，导致过拟合。解决办法就是限制最大深度。

要点3：CatBoost主要采用对称树的生成形式，而LightGBM和XGBoost的生成形式依旧是传统的决策树。在对称树中，每一层每一个节点的判断条件是一样的，而在传统决策树里面，每一层的判断条件是不同的。

GBDT的DART是怎么用？XGBoost里面的某个（记不清了，很冷门）API是怎么用的？

要点：这题目给我问懵逼了，没听过DART是啥好吗...完事查了才知道这东西全称**Dropouts meet Multiple Additive Regression Trees**，利用了深度神经网络中dropout设置的技巧，随机丢弃生成的决策树，然后再从剩下的决策树集中迭代优化提升树，这是DART的主要思想。

三种决策树是如何处理缺失值的（划分节点时-训练模型时-预测时）

id3无法处理缺失值，c4.5引入带权的概念处理，cart则使用插值的方法或者代理分裂；

54 决策树防止过拟合的方法（除了减枝，还有bootstrap，DBA，貌似说的这两，没听清楚）

1.预剪枝和后剪枝；

2.集成；

8、介绍一下决策树可能出现的问题及相应的解决办法

在决策树学习的过程中，为了尽可能正确分类样本，节点划分过程不断重复，有时会造成决策树分支过多，以致于把训练集自身的一些特点当做所有数据都具有的一般性质而导致过拟合。

解决方法：剪枝（包括预剪枝和后剪枝）

gbdt为什么用梯度，用梯度有什么好处，为什么GBDT用负梯度当做残差

gbdt的原理，写出推导过程，每一步都停下来问两句。gbdt的梯度提升体现在什么地方，在损失函数不是平方损失函数时为什么可以用t-1步分类函数相对于损失函数的负梯度来作为残差的近似值，数学意义。

gbdt中每个弱分类器是什么，为什么用cart回归树而不是分类树，节点的分裂规则，写出公式。公式中的每个值在实际训练过程中是怎么计算出来的，举例子说明

1. 此处打断项目，让我说一下gbdt的训练过程

1. gbdt的训练过程，做分类预测时候如何训练的，cart回归树的节点分裂过程。对于连续或离散变量，树节点如何划分类别

2. gbdt的概率是怎么计算的？gbdt能否计算多分类问题，是互斥多分类问题，还是非互斥多分类？如果要做非互斥多分类，在gbdt哪一步做改进？

3. 残差是怎么计算的，损失函数的形式

1、Adaboost详细，adaboost的权值和RF比较

- Xgb里防止过拟合可以设置的参数
- GBDT 原理讲一下。追问：负梯度的概念是什么，为什么用负梯度
- Xgboost和GBDT 叶节点权重怎么算

$$w_j^* = -\frac{\sum_{i \in I_j} y_i}{\sum_{i \in I_j} h_i + \lambda} \quad (\text{式10})$$

- 这是xgboost，而gbdt下面是一个常数1
-
- GBDT 单调性约束怎么实现的
- 能否用LR 或者 SVM 做GBM基模型：
- 理论上可以实现，但是计算复杂度太高了，特别是svm，进行大量的集成非常耗费时间；并且二者的多样性相对于tree来说差多了；
- RF 原理讲一下，为什么能防止过拟合
-
- LightGBM原理，追问：直方图算法的具体实现逻辑（是否看过源码）
-

介绍下xgb是如何调参的，哪一个先调，哪一个后调，为什么？哪几个单独调，哪几个放在一组调，为什么？哪些是处理过拟合的，哪些是增加模型复杂程度的，为什么？

1. xgb是如何实现并行的。

保存预排序的block，用进程间的通信并行寻找最优分裂点。

4.lgb的直方图优化算法说说。

- 5.假设信息增益函数 $\text{entropy}(x,y)$,实现特征重要性的计算
`featureImportce(feature,label)`(写代码)

lightgbm 为什么 更快

算法优化和工程优化两个层面来达到更好的性能，具体可见原理部分；

**lightGBM, XGBoost, GBDT怎么分裂 怎么
着最优找分裂点（xgboost如何选择最优分
割点）**

lightgbm创建每棵树时速度是均匀的吗？

不是，lgb的每棵树的建树过程和当前的数据子集以及上一轮基模型拟合得到的负梯度等都有关系，可能有的树建3层可能有的6层等，因此速度一般不是均匀的；

**lightgbm训练时和特征数目更相关还是样本数目更相关
都相关。。。什么鬼问题**

连续特征值在lightgbm中如何找到分界点
具体可见上面的原理部分；

gbdt如果有100棵树，每棵树的输出是什么？

当前拟合结果和上一轮负梯度经过损失函数得到的负梯度；

4、xgboost如何输出概率？ 算法上怎么输出概率的？

通过sigmoid函数将结果压缩到0~1区间输出概率；

7、xgboost gblinear和gbtree的区别

8、xgboost min_child_weight为什么可以防止过拟合？

`min_child_weight : int`

Minimum sum of instance weight(hessian) needed in a child

也就是叶子节点分母部分的二阶导，控制最小二阶导，如果分母部分太小则整

个叶子节点的权重太大， $w_j^* = -\frac{\sum_{i \in I_j} r_i}{\sum_{i \in I_j} h_i + \lambda}$ ，叶节点权重过大，会导致最终的结果受到部分叶节点的影响较大，影响泛化性能；

假设xgboost有10个叶子节点，每个叶节点的权重结果为：

$$w_j^* = -\frac{\sum_{i \in I_j} r_i}{\sum_{i \in I_j} h_i + \lambda}$$

这里的 w_i 和逻辑回归中的 w_i 的概念很类似，以二分类问题为例，xgboost的最终输出为：

$$w_1 * I + w_2 * I + \dots w_{10} * I$$

这里的 I 不是1就是0，1表示这个样本落入这个叶子节点，0表示没有落入，则最终的输出就是

$$\text{sigmoid}(w_1 * I + w_2 * I + \dots w_{10} * I)$$

和 l_1 一毛一样，因此我们使用`min_child_weight`实际上就是限制 w 不能太大，这和逻辑回归中通过 l_1 限制权重 w 的大小是基本类似的思路；

1. 使用GBDT的时候一些参数（比如树的深度、树的颗数等）是怎么调的？
2. • 树模型（非线性模型）和线性模型的区别？树模型的优势是什么？
3. • 让我写一下信息增益、信息增益率、基尼系数三个公式。

1. 树模型对连续型 离散型特征有一套统一的理论是什么？
2. xgboost和GBDT的分裂方式你认为哪个好。

Xgb的更好，起到预剪枝的作用，而xgb之前的gbdt用cart tree需要人工设定参数进行预剪枝；

3. GBDT的特征组合原理

Gbdt每一步分裂实际上都是一个if else的规则，n步分裂则会形成n个if else的规则，这n个if else的规则实际上是对原始的特征矩阵中的特征进行特征组合选择其中最有意义的特征

组合，这一点和nn进行表征学习的思路是非常类似的。

4. 决策树相比其他算法有什么优势?

- 1.拟合能力强;
- 2.可解释性强;
- 3.算法构建很简单不需要涉及到lr那样麻烦的迭代;

2、ID3\C4.5等基本树 是二叉树还是多叉

树 被切过的特征还会再切吗

离散特征(离散数量>2)是多叉分类，连续是二叉分裂，连续可以在切，离散不可以（当时回答的是可以再切，被提示后意识到不可再切，说了自己的matlab实现，先做集合，遍历特征，保存最大的信息增益位置，然后对特征切分，切分后把这个特征从集合中删掉，所以离散特征切完就不在切了，还好反应过来了，连续性特征可以再切，详情可以去看看别人的ID3树和其他树的源代码)

XGBOOST损失函数/正则怎么算

见原理部分

信息熵公式，说说联合熵的公式

下.

联合熵: $H(x, y) = - \sum_{i=1}^n p(x_i, y_i) \log p(x_i, y_i)$

条件熵: $H(y|x) = \sum_{i=1}^n p_i H(y|x = x_i)$

信息熵: $-\sum_{i=1}^n p_i \log p_i$

联合熵=信息熵+条件熵

xgb二阶泰勒展开，为什么不三阶？

CART树和ID3区别？ID3的缺点？ID3换成基尼系数是否就没有这个缺点了？

区别见上，缺点一样，**gini**和信息熵都会偏向取值较多的特征；

CART回归树和ID3是二叉树还是多叉树？

一个二叉树一个是多叉树